NORTHEASTERN UNIVERSITY

MASTERS THESIS

USBeSafe: Applying One-Class SVM for Effective USB Event Anomaly Detection

Author:
Brandon L. DALEY

Supervisor:
Dr. William ROBERTSON

A thesis submitted in fulfillment of the requirements for the degree of Master of Science in Computer Science

in the

Northeastern University
College of Computer and Information Systems

April 25, 2016

This material is based upon work supported under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001.
Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the U.S. Air Force.

Declaration of Authorship

I, Brandon L. DALEY, declare that this thesis titled, "USBeSafe: Applying One-Class SVM for Effective USB Event Anomaly Detection" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at Northeastern University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.
- The views expressed in this thesis do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government.

Signed:

Date:

"When we contemplate the whole globe as one great dewdrop, striped and dotted with continents and islands, flying through space with other stars all singing and shining together as one, the whole universe appears as an infinite storm of beauty."

John Muir, *Travels in Alaska*, 1915

NORTHEASTERN UNIVERSITY

Abstract

College of Computer and Information Systems

Master of Science in Computer Science

USBeSafe: Applying One-Class SVM for Effective USB Event Anomaly Detection

by Brandon L. DALEY

Increased use of transient devices such as wireless keyboards, webcams, and flash storage in the last ten years has drastically increased the surface area on which attackers can target vulnerable systems. USB devices, a subclass of transient devices (TDs), have become a common transport mechanism for malware making its way into a target machine or network. The rogue-TD attack class, demonstrated by BadUSB, relies on updating the device firmware to perform malicious actions and can be undetectable at the end-user level if written effectively, as the attack hides in plain sight.

In this thesis, we present *USBeSafe* as a first-of-its-kind machine learning-based anomaly detection framework for detecting a specific subclass of rogue-TD attack in which a covert keyboard interface is defined on a seemingly benign device. We apply machine learning techniques, specifically one-class support vector machines, to create an offline USB event anomaly detection system that serves as the basis for a live detection system. The USBeSafe system provides an extensible framework for efficient USB traffic feature extraction, model selection and training, and classification.

We examine a wide array of attributes that factor into model prediction performance such as USB traffic feature types, contextual information via n-grams, and model kernel function with associated parameters. We then apply them to a search for ideal attributes in classifying benign USB keyboard traffic with an input corpus collected over eight months. Using viable candidates from this search, we train 51 models and test them against a known malicious rogue-TD covert keyboard attack. Through these results, we provide an analysis of feature and model attribute relevance specific to benign USB keyboard traffic and the basis for a live USBeSafe system.

We find that, while there exists little correlation between novel classification score and kernel function used, incorporating USB packet interarrival times and larger n-grams generally increase the novel observation score against the malicious input. We also find that, as expected, more information yields higher scoring, with the highest performing model against the attack sample being trained on 2-grams of all possible features considered: packet interarrival times, types, and payloads.

Acknowledgements

First and foremost, I want to express my sincere gratitude to my advisor, Dr. William Robertson. He is an extremely busy man and somehow still manages to take on an M.S. student for a thesis. His brilliance has opened my eyes to so many different avenues for research and has helped guide this thesis towards truly meaningful contributions to the field. For his time, knowledge, and wisdom, I thank him.

My co-advisor, Graham Baker, has been a constant guiding force during my fellowship at MIT Lincoln Laboratory. For two years, he has helped refine many aspects of my academic and research abilities. I greatly appreciate the time he has taken in helping develop not only this research, but me as well.

I would like to extend my thanks to the entirety of Group 59, Cyber Systems Assessments, at MIT Lincoln Laboratory. The group's consistent intellectual and financial support to make this thesis and degree possible cannot go unmentioned.

I am deeply grateful for the time the U.S. Air Force has allowed me to pursue this degree. I cannot express enough how much appreciation I have for an organization that has avenues for its Airmen to academically and professionally flourish.

I thank Capt Chris Patterson, U.S. Air Force, for his brilliant eye when it comes to editing, ability to question everything, and ideas when I was in a rut. I appreciate the time he chose to devote listening to me hash out issues, providing suggestions, and helping make this thesis shine.

Above all except my God, I want to thank my family and friends for their love, support, and patience over the last two years. My grandparents, to whom this thesis is dedicated, have put up with numerous discussions outside their areas of expertise, through the struggles and the breakthroughs of this research.

Contents

Declaration of Authorship iii

Abstract vii

Acknowledgements ix

Contents xi

List of Figures xv

List of Tables xvii

List of Abbreviations xix

1 Introduction 1
- 1.1 TD-based Security Threats . . . 1
- 1.2 Motivation: Solving the "Rogue TD" Problem . . . 2
- 1.3 Focus and Contributions . . . 3

2 Background and Related Works 5
- 2.1 The Universal Serial Bus . . . 5
  - 2.1.1 History and Overview . . . 5
  - 2.1.2 Benefits of USB . . . 5
  - 2.1.3 How does the USB Protocol Work? . . . 6
    - Endpoints and Transactions . . . 6
    - Transfer Types . . . 7
    - Enumeration: Learning about the Device . . . 8
- 2.2 The Science of Machine Learning . . . 9
  - 2.2.1 Raw Data vs. Features . . . 10
    - An Example Feature: n-grams . . . 10
    - Feature Selection . . . 11
  - 2.2.2 Categorization: Learning vs. Output . . . 11
  - 2.2.3 Classification Problems . . . 12
    - Support Vector Machines . . . 12
    - OCSVM Specifications and Parameters . . . 13
  - 2.2.4 Model Fitting . . . 14
    - Parameter Estimation . . . 15
    - k-Fold Cross Validation . . . 15
- 2.3 Detecting Malicious Activity . . . 16
  - 2.3.1 Anomaly Detection . . . 16
    - SVM-based NAD Systems . . . 18
- 2.4 Review of USB-based Attack Vectors . . . 18
  - 2.4.1 Oops... I Dropped It . . . 18
  - 2.4.2 Autorun.inf . . . 19
- 2.5 BadUSB – A Novel Type of Attack . . . 19

2.5.1 Existing Defenses and Limitations 20
IEEE 1667 21
Linux and the GoodUSB Response 22
Windows Patch 3143142 24
2.5.2 Enter: USBeSafe 24

3 Formalization and Implementation **25**
3.1 Threat Modeling 25
3.2 System Overview 26
3.3 Data Collection 26
3.3.1 `usbmon` 26
3.3.2 Traffic Capture 27
3.3.3 Understanding the Data 27
3.4 Feature Extraction 28
3.4.1 Data Preprocessing 28
3.4.2 Potential Feature Selection and Extraction 30
Packet Interarrival Times 30
Packet Type 31
Packet Payload 31
Extraction and Storage 31
3.5 Feature Preprocessing 32
3.5.1 Scaling 32
3.5.2 n-grams 32
3.6 Model Searching 33
3.6.1 Framing the Search Space 33
3.6.2 Search Algorithm 33
3.7 Model Training 34
3.8 Model Testing 35

4 Experimentation and Results **37**
4.1 Data Collection and Feature Generation 37
4.1.1 Expectation vs. Reality 37
4.2 Model Searching 38
4.2.1 Performing the Search 38
n-gram Possibilities 38
Features Selection 39
Parameter Settings 39
k-Fold CV 39
Scoring Mechanisms 40
4.2.2 Search Experiment Framework 40
4.2.3 Experiment Results and Discussion 41
4.3 Model Training and Testing 44
4.3.1 Model Training Experiment 44
4.3.2 Model Testing Experiment Framework 44
4.3.3 Model Testing Results 44

5 Conclusions and Future Work **49**
5.1 Conclusions 49
5.2 Future Work 50

A USB Descriptors **51**

B Interarrival Time Histograms by `pause` Length **53**

C Model Training Requests **59**

D USBeSafe Project Files **63**
D.1 Project Hierarchy 63
D.2 Directory and File Descriptions 64

Bibliography **69**

List of Figures

2.1 USB Descriptor Hierarchy 9
2.2 k-Fold CV in Action 16
2.3 Covert Keyboard Interface Enumeration 21
2.4 GoodUSB Infrastructure 23
2.5 GoodUSB Operation 23

3.1 USBeSafe System Flow 26
3.2 `usbmon` Output Wireshark Sample 27
3.3 USBeSafe Data Preprocessing Hierarchy 29

4.1 Model Search Results based on Independent Attributes . . . 41
4.2 Malicious Test Results based on Independent Attributes . . . 45
4.3 Novel Observation Score Distribution by Kernel Function . . 46
4.4 Comparing Model Search Accuracy Scores and Test Novel Observation Scores 47

B.1 Class 0x00 20000ms-`pause`, 200ms Interval Itime Histogram 53
B.2 Class 0x00 20000ms-`pause`, 500ms Interval Itime Histogram 53
B.3 Class 0x00 40000ms-`pause`, 200ms Interval Itime Histogram 54
B.4 Class 0x00 40000ms-`pause`, 500ms Interval Itime Histogram 54
B.5 Class 0x00 60000ms-`pause`, 200ms Interval Itime Histogram 54
B.6 Class 0x00 60000ms-`pause`, 500ms Interval Itime Histogram 55
B.7 Class 0x03 20000ms-`pause`, 200ms Interval Itime Histogram 55
B.8 Class 0x03 20000ms-`pause`, 500ms Interval Itime Histogram 55
B.9 Class 0x03 40000ms-`pause`, 200ms Interval Itime Histogram 56
B.10 Class 0x03 40000ms-`pause`, 500ms Interval Itime Histogram 56
B.11 Class 0x03 60000ms-`pause`, 200ms Interval Itime Histogram 56
B.12 Class 0x03 60000ms-`pause`, 500ms Interval Itime Histogram 57

List of Tables

2.1 Comparison of USB 2.0 and 3.x Generations 6
2.2 Properties of USB Transfer Types 7
2.3 Common USB Device Class Codes 9
2.4 Overview of Classification Errors 15

3.1 OCSVM Data Input Structure 33

4.1 OCSVM Attribute Contributions to Search Space 40
4.2 Attribute Effects on OCSVM Accuracy Scores 42
4.3 Incomplete Search Tests 43
4.4 Feature Subset to Novel Observation Score Correlation . . . 46

A.1 Device Descriptor Contents 51
A.2 Configuration Descriptor Contents 51
A.3 Interface Descriptor Contents 51
A.4 Endpoint Descriptor Contents 52
A.5 String Descriptor Contents 52

List of Abbreviations

ACK	**Ack**nowledgment
ANN	**A**rtificial **N**eural **N**etwork
CD	**C**ompact **D**isc
CD-ROM	**C**ompact **D**isc-**R**ead **O**nly **M**emory
CV	**C**ross **V**alidation
DARPA	**D**efense **A**dvanced **R**esearch **P**rojects **A**gency
DoD	**D**epartment **o**f **D**efense
DHCP	**D**ynamic **H**ost **C**onfiguration **P**rotocol
FP	**F**alse **P**ositive
HID	**H**uman **I**nterface **D**evice
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
INS	**I**ndian **N**aval **S**hip
I/O	**I**nput/**O**utput
ML	**M**achine **L**earning
mRMR	**m**inimum-**R**edundancy-**M**aximum-**R**elevance
NAD	**N**etwork **A**nomaly **D**etection
NIC	**N**etwork **I**nterface **C**ard
OCSVM	**O**ne-**C**lass **S**upport **V**ector **M**achine
PCAP	**Pa**cket **C**apture
RBF	**R**adial **B**asis **F**unction
SVM	**S**upport **V**ector **M**achine
TD	**T**ransient **D**evice
URB	**U**SB **R**equest **B**lock
USB	**U**niversal **S**erial **B**us

Dedicated to my grandparents: without your love and devotion in raising me, none of this would be possible. From the depths of my heart, thank you for every opportunity you have ever allowed me. I love you both, and I hope you can be proud.

Chapter 1

Introduction

In today's information age, individuals possess the ability to use removable media devices to store gigabytes of information in their pockets and transport that data to other parts of cyberspace with relative ease. With the explosion of on-the-go computing and data storage in the last decade, transient devices (TDs) such as flash drives, external disk drives, wireless keyboards, web cameras, and numerous others have become commonplace in end-user computing. Even with the massive increase in cloud computing and storage as of late, the utilization of TDs remains steady; the newest major hardware iteration of USB, version 3.0 or SuperSpeed, is expected to be shipped inside three billion devices by 2018 [1]. Every one of these devices contains a small micro-controller that manages interactions between a host machine and the device itself. These micro-controllers negotiate connections with hosts and transfer data packets as necessary for operation. Without appropriate security measures in place, malicious takeover of these microcontrollers can prove a serious threat.

1.1 TD-based Security Threats

As with any cyber technology, it was only a short time before malicious actors worked to exploit vulnerabilities in TD mechanisms; many instances of security breaches in recent years illustrate that hackers have taken advantage of the inherent TD transport mechanism, namely humans, to successfully spread malware, take control of systems, and exfiltrate valuable information.

In 2008, after a piece of malware dubbed *Agent.btz* began propagating via flash drives and CDs across U.S. Department of Defense (DoD) networks, all forms of removable media within the DoD were banned indefinitely. *Agent.btz*, though of a publicly unknown origin, served as both a beaconing and exfiltration agent. This security breach eventually led to one of the largest modern-day reformations in the U.S. military with the standup of United States Cyber Command, a major military command dedicated to the defense and stability of U.S. cyberspace resources [2].

Then, in 2010, the world witnessed the first known instance of a cyber attack causing physical destruction – Stuxnet. Known as Olympic Games inside U.S. intelligence circles, Stuxnet, after months of sophisticated infrastructure mapping, quietly damaged a significant portion of uranium enrichment centrifuges at the Iranian nuclear facility in Natanz, all while reporting to engineers and maintainers that systems were running normally. Estimating that the virus set Iran's nuclear program back anywhere from twelve months to five years, Stuxnet only made it inside the facility in the

first place because of a rogue USB thumb drive. Left in the parking lot at Natanz and unbeknownst to the worker that picked it off the ground, that thumb drive held a dangerous piece of malware capable of crippling Iranian uranium enrichment [3, 4].

In 2012, the Indian navy suffered a blow to its cyber security posture when it was found that a Chinese crawler made its way into the air-gapped network of the INS Arihant, India's first nuclear submarine. The malware cached documents in the networks of the submarine that matched specific keywords, waiting to perform covert data transfer back to Chinese IP addresses until the flash drive it resided on was connected to any computer with internet access [5].

On a macro-scale, the security issues TDs cause are countless. One study performed in 2011 found that, in only the two year span prior, 50% of organizations, both public and private, had sensitive information on USB drives compromised [6]. These attacks and data compromises rely on users who, unknowingly or not, act as malware carriers by transferring infected TDs across hosts, networks, and physical boundaries.

Most recently, a new class of attacks introduced by researchers of BadUSB takes these malicious actions one important step further: building them directly into TD firmware by reprogramming their microcontrollers, allowing them to masquerade as benign devices [7, 8]. BadUSB shifts the attack paradigm tremendously with regards to TDs; it demonstrates that a device itself can be inherently malicious, not just due to code stored on a *properly* functioning device [7]. A device that functions properly is one that performs operations in the benign way that a human user expects of it. This shift caused by BadUSB means that authors can more easily hide their malware in plain sight, making it of great value to identify potentially malicious behavior before a host-based device driver acts on incoming USB traffic.

1.2 Motivation: Solving the "Rogue TD" Problem

Since TDs have gained mainstream use starting in the early 2000s, attack vectors via these devices have become increasingly sophisticated. Because of attacks like those demonstrated with BadUSB, which we call *rogue-TD attacks*, it is now even easier for malicious code to hide in plain sight. We see that it is now possible for an actor to reverse engineer the existing firmware on a TD, rewrite that firmware to perform some malicious action while preserving the operational intent of the TD, and reflash that TD with the modified firmware in the hopes that he can get it into the hands of an unknowing person who can facilitate pairing the TD with a host machine.

It is straightforward to imagine a scenario in which this type of attack could bypass some security measures to gain access to a machine and possibly a complete network. Like we saw it work at Natanz [3, 4], an adversary A leaves a flash drive D in the parking lot of the target organization. Employee E at the organization picks up D and carries it into the building past any security checkpoints; at this point, A has gained physical access to the organization. Even if E does scan the device at a security kiosk for malicious files, these machines typically only scan storage partitions of a device, meaning the firmware goes unchecked and the report likely comes back with no red flags. E plugs the device into his work computer; now A

has access to E's machine, though W is unaware. E finds nothing stored on the drive and decides to leave it in his machine for extra storage. E password-locks his machine and, while he goes to get his morning coffee, the screensaver on his machine activates. The malicious firmware has been monitoring for this event and activates a shell; uses the password, stored by a keylogger when E locked the host, to gain root access; and begins exfiltrating files to a remote machine in A's control.

This attack scenario is just one example of what is possible when a maliciously provisioned device gains access to a target host machine. While existing solutions to the rogue-TD attack paradigm require much in the way of access control maintenance and certificate management or a user-defined policy infrastructure that substantially increases user workload (see section 2.5.1), we seek to find a way to keep device enumeration and traffic monitoring completely in the background by relying on a host-based detection approach of malicious USB events. Such a solution allows flexibility for: (1) organizations to use standard devices, (2) manufacturers to avoid changing how their hardware operate, and (3) users to continue using TDs according to the *status quo*.

1.3 Focus and Contributions

In this research, we introduce USBeSafe as a means of detecting forms of a specific kind of rogue-TD attack: a keyboard emulation attack when a covert human interface configuration is defined in the device firmware, like that demonstrated by BadUSB [8]. USBeSafe leverages machine learning models that are taught against benign USB traffic. This research provides a number of novel contributions to the field of USB security when examined under the lens of rogue-TD attacks:

1. Creation and characterization of a USB traffic corpus containing USB packets as they travel across a bus, utilizing the semantics of the Linux kernel module `usbmon`.

2. Constructed extensible framework for and efficient extraction of identifying features from USB traffic, with focus on USB protocol level semantics.

3. Novel application of one-class support vector machines to accurately identify patterns of USB traffic specific to identified USB device classes.

4. Performed extensive feature and model attribute selection for USB keyboard traffic with analysis of relevance specific to the USB traffic corpus against trained models.

5. Built platform for applying one-class support vector machines to both offline and real-time USB traffic anomaly detection systems.

6. Identification of potential future work to further the effectiveness of USB event anomaly detection.

Chapter 2

Background and Related Works

In this chapter, we discuss a number of background topics to gain an understanding for the foundation upon which USBeSafe is built. We first cover the fundamentals of USB communication, an overview of machine learning, to include methods and goals, and the development of anomaly detection mechanisms. We then present a history of USB-based attacks, including the most recent and lethal, BadUSB, along with associated protection mechanisms. These subjects lay the groundwork for USBeSafe, a machine-learning based USB event anomaly detector.

2.1 The Universal Serial Bus

2.1.1 History and Overview

The Universal Serial Bus (USB) specification and protocol was first introduced in 1996 on a very small scale with generation 1.0, meant to act as a versatile interface for a large number of external devices. Over the years, a number of revisions have been made to the specification, including USB 1.1, 2.0, 3.0, and 3.1, all focused on increased data transfer rates between device and host and device power management [9].

Currently, USB 2.0 and 3.x devices are widely used to facilitate a large number of device-to-host connections. Table 2.1 illustrates some fundamental differences among the USB 2.0 and 3.x specifications [10]. After developers realized the potential for the USB standard, USB 2.0 was created to support a broad range of devices that require higher current as well as faster bus speeds.

Recently, largely due to inflated file sizes from high-definition audio and video as well as the common desire for end-users to transfer massive amounts of data from one piece of hardware to another, the USB 3.0 standard was introduced. The third major generation of USB facilitates significantly faster data transfer rates, utilizing a two-bus architecture that can run in parallel [11]. Due to this new architecture, USB 3.0 devices can support data transfers in both directions simultaneously.

2.1.2 Benefits of USB

The creators of the USB standard set out to introduce a specification that provides versatility and extensibility for many different computing needs. USB serves as a stable platform for a number of TD-to-host connections for users as well as for hardware developers.

On the end-user side, there are a number of benefits that come from utilizing USB [9]:

TABLE 2.1: Comparison of USB 2.0, 3.0, and 3.1 features [10].

Generation	*USB 2.0*	*USB 3.0*	*USB 3.1*
Reverse Compatible	USB 1.1	USB 1.1/2.0	USB 1.1/2.0/3.0
Max. Transfer Rate	480Mbps	5Gbps	10Gbps
Charging Power	100 mA	900 mA	900 mA
Year Introduced	2001	2009	2014
Popular Usage	Mass storage, HID devices, printers	Mass storage	Mass storage

- **Ease of use:** one physical interface for a large number of peripheral operations, hot-pluggable ports, straightforward physical connections, automatic device configuration with appropriate drivers
- **Cost:** hardware components for USB devices are inexpensive, expanding possibilities for end-user computing
- **Reverse compatibility:** possible to use older generation devices on newer generation buses, so users do not lose existing capabilities as USB gets faster and more powerful

For developers looking to leverage USB's capabilities, its benefits are rather similar. USB's versatile nature allows a continually growing spectrum of devices to conform to its specifications, and the vast majority of operating systems already have native support for device recognition and enumeration.

2.1.3 How does the USB Protocol Work?

The infrastructure in place to support USB is actually extremely complex; fortunately for users, managing bus-level interactions between a USB device and a host machine is performed automatically by the host, barring the infrequent case where a user must manually install device drivers.

Throughout this section, the discussion of USB functionality is geared towards USB 2.0 capable devices, arguably most numerous today. While there are some differences with USB 3.0, they are minor and lend little to further understanding of how the protocol operates.

Endpoints and Transactions

Each physical bus on a host machine is capable of managing transactions of up to 127 USB devices. USB is a host-centric protocol, meaning that the host initiates all transactions with the USB device [12]. All transactions via USB occur at *endpoints* on the device that act as data sources and sinks. When a device is enumerated by the host, both IN and OUT endpoints are addressed for data flow. The IN endpoint stores data coming *in* to the host, and the OUT endpoint receives data going *out* from the host [9].

Physically, these endpoints are buffers or registers in the device, facilitating USB transactions, of which there are three: (1) IN, (2) OUT, and (3) Setup. During an IN transaction, data flows from the device to the host,

TABLE 2.2: USB transfer types and their various identifying characteristics [9].

Transfer Type	*Control*	*Bulk*	*Interrupt*	*Isochronous*
Usage	identification, configuration	printer, scanner, drive	mouse, keyboard	audio, video
Transaction Types Used	setup	all	all	all
Data Flow Direction	IN and OUT	IN or OUT	IN or OUT (IN for USB 1.0)	IN or OUT

while the opposite is true of an OUT transaction. A Setup transactions is a special-case OUT transaction in which the host forces the device to respond to the Setup request. Each USB transaction consists of a series of packets [9]:

- **Token packet:** host-generated; contains upcoming transaction type (IN/OUT/setup), as well as device and endpoint addresses
- **Data packet:** optional, and contains any necessary payload
- **Status packet:** acts as a handshake or error reporting mechanism

Transfer Types

Every USB transaction is defined at a high level by the type of data transfer either the host or the device initiates. USB supports four transfer types, all suited for different purposes [9]:

1. **Control transfers:** utilized when (a) the host initiates a data request from the device to learn about and configure the device for use on the host, and (b) device vendor- or class-specific requests are defined.
2. **Bulk transfers:** fastest type of transfer; used for sending large amounts on non-time-critical data; when bandwidth is low on the bus, bulk transfers have the lowest priority.
3. **Interrupt transfers:** used for devices that require a fast response; under bandwidth constraints, the host gives priority to these types of transfers because there exists a guaranteed maximum latency for response.
4. **Isochronous transfers:** streaming transfers for when delivery rate is important; ensures some amount of reserved bandwidth; no retransmission of data received with errors.

Important properties and uses of these transfer types can be found in Table 2.2 [9]. Note the special case where USB 1.0 devices support only IN transactions for interrupt transfers. Many human-interface devices (HIDs) only require low-speed data transmission, meaning they can run on USB

1.0 hardware. HID devices also tend to be interrupt-based systems, where the device sends an interrupt to the host, indicating that it has data for the host. We can understand this most easily when considering a keyboard or mouse. The OS does not foresee that a user will type a character or move the mouse to the left; an interrupt transfer occurs, originating from the IN endpoint of the device. For HID devices, interrupt transfers do not occur with an OUT transaction; HID devices serve to provide interrupts rather than receive them.

Enumeration: Learning about the Device

For a host to properly manage traffic coming from a specific device, it needs to: address the device on the bus and its endpoints; read device-provided descriptors containing information on device functionality; and load a driver based on configuration information. To accomplish this, the host initiates a series of control transfers.

The OS manages all steps of enumeration behind the scenes, hidden from the user, except for end-state indications whether the device was configured correctly or if the process resulted in error. The enumeration process, while technically complicated, can be broken down into a few major steps:

1. First, a user introduces a new USB device to the system, and the hub detects a voltage change in one of its power lines, indicating a device is now powering via the hub [9, 12].

2. The hub determines speed capabilities of the device (low/full/high) and prepares the device to receive and respond to control transfers from the host via a default device address and a default endpoint, both 0x00 [9, 12].

3. The host sends an initial Get Descriptor request to the device, asking for the maximum packet size via the default endpoint supported by the device. The device will send a Get Descriptor response with the requested data, and the host provides an address on the bus to the device [9, 12].

4. After the device has been uniquely addressed, the host sends another Get Descriptor request to the device, receiving the entire device descriptor in response, containing device information and number of configurations.

5. For all configurations on the device, the host will send as many Get Descriptor requests as necessary to retrieve information from the device on all configurations the device implements. In turn, the host will send further Get Descriptor requests for subordinate descriptors, if available (interface descriptors, then endpoint descriptors).

6. The host loads a device driver to use for further communication with the device, and the driver can send a Set Configuration request for the device to operate under a specific configuration [9].

Figure 2.1 illustrates the descriptor hierarchy that the host learns about during the enumeration stage [12]. All USB devices must provide a single

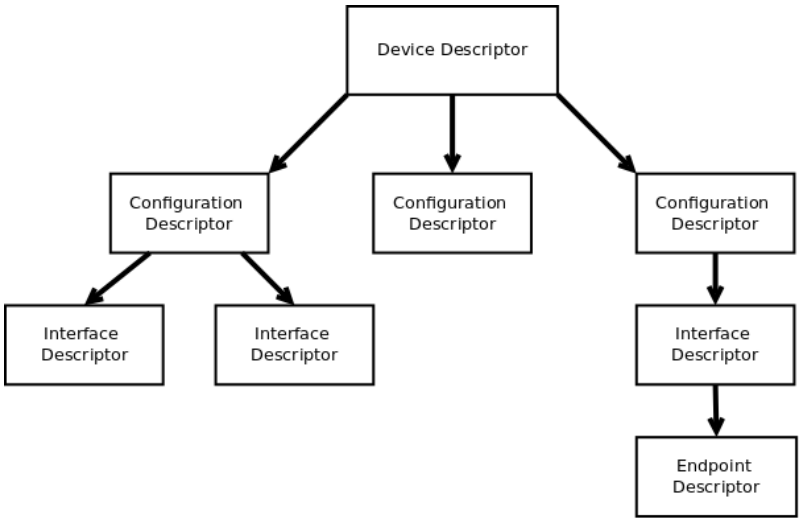

FIGURE 2.1: The descriptor hierarchy of a USB device, indicating different ways the device can provide the main descriptors.

device descriptor, along with one or more configuration descriptors. Note that there are a number of other descriptor types not discussed here that contain further information; Appendix A contains a table for each descriptor type, identifying the data that resides in the Get Descriptor responses.

USB device class codes can be located in device and interface descriptors and help the host determine the proper driver to load. A summary of common USB class codes can be found in Table 2.3 [13].

TABLE 2.3: Common USB class codes based on what descriptors in which they reside [13].

Class Code	Descriptor	Description	Examples
0x00	Device	Unspecified	Code found in interface descriptor
0x01	Interface	Audio	Speaker, microphone
0x03	Interface	HID	Keyboard, mouse
0x06	Interface	Image	Webcam, scanner
0x07	Interface	Printer	Inkjet or laser printer
0x08	Interface	Mass storage	USB flash drive, external drive
0x09	Device	USB hub	Multi-device USB hub
0x10	Interface	Audio/Video	Webcam, television

2.2 The Science of Machine Learning

Machine learning (ML), with respect to the field of computer science, was first defined by Arthur Samuel in 1959 as a "field of study that gives computers the ability to learn without being explicitly programmed" [14]. Later, Mitchell formalized the definition of machine learning: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E" [15]. Today, machine learning

involves the use of algorithms that take training data as input for learning generalizations and can make predictions about new input based on those generalizations [16, 17].

Over the years, ML has gained prevalence in many applications, including, but not limited to:

- Sentiment prediction [18]
- Spam filtering [19]
- Search engine optimization [20]
- Character and voice recognition [21, 22]
- Predictive text [23]

2.2.1 Raw Data vs. Features

Generally, ML models do not accept raw data for training or testing. During development, a model must be trained on *features* that effectively characterize the raw data. During model learning, data is often provided as a matrix of shape (n, p), where n is the number of samples or observations and p is the number of features types used to represent the data. This matrix can be represented as a list of *feature vectors*, where each vector is a list of length n of feature values over all observations.

An Example Feature: n-grams

Consider the case of feature selection for a predictive text model. Predictive text is typically based on Markov chains [24], models that calculate the probability of transition to a future state given a current state. Model training and predictions can rely on *n-grams*, a contiguous sequence of n tokens in the data, where the concept of a token is defined in the application. For example, consider the sentence:

"The dog is fat and brown"

The list of bigrams (n = 2) with tokens defined as individual words delimited by space characters would be:

["The dog", "dog is", "is fat", "fat and", "and brown"]

These n-grams can then, for example, help define the Markov model states, and, after enough training data has been collected and meaningful probabilities generated for state transitions, the predictive text model can provide likely suggestions for future text based on current inputs. N-grams prove rather useful because they can reduce information loss. Instead of a simple one-word unigram, a larger n can provide more context for a current state, leading to more accurate future predictions.

Feature Selection

As expected, selecting features from raw data is a difficult subproblem in any ML implementation. Much work has been done in the field of feature selection to provide insight into selecting relevant features based on raw data.

The central premise of feature selection is to remove irrelevant and redundant features from a list of possible features [25, 26]. More formally, P is the set of possible features, feature selection techniques work to define S, where $S \subset P$.

Feature selection algorithms take on many different forms, to include:

- *Subset selection* implements a search through possible subsets of features, finding the subset with the best score based on a defined score metric. During subset selection, exhaustive search is computationally expensive; therefore, greedy algorithms are typically used to achieve some previously defined score threshold or until no improvement is made [27].

- *Minimum-redundancy-maximum-relevance*, or *mRMR*, proposed by Peng, *et al*. [28] uses mutual information, correlation, or similarity scores as metrics to determine features. In mRMR, a feature's relevance score is decreased when it has a high redundancy score in the presence of other features. If the relevance score is low enough, the feature will be removed from the target S.

- *Correlation feature selection* (*CFS*) is based on the idea that an effective S will contain features that are highly correlated with the label and have low correlation to one another [29].

2.2.2 Categorization: Learning vs. Output

It is useful to categorize the generation and use of ML models in two ways based on: (1) the information available for learning, and (2) the type of result as output.

Categorization based on learning typically lends itself to two methods: *supervised* and *unsupervised* [30, 31]. When an ML model learns in a supervised context, labels acting as desired output are supplied to the ML algorithm for each input supplied. The goal of supervised learning is to make generalizations that map inputs to one or more supplied desired outputs. Unsupervised learning, on the other hand, does not rely on labels. ML algorithms that generate models via unsupervised learning must infer structure, generalizations, and rules from input alone, with no labels provided.

Categorization based on output deals with the form of the desired result from an ML system. Major output types include:

- **Classification:** the ML algorithm divides input data into one or more classes, generating a model that attempts to predict the class of new inputs; typically a supervised tasks where the ML algorithm receives labels corresponding to classes during training [30–32]

- **Regression:** similar to classification problems, except that the outputs are continuous variables and not distinct classes; like classification, regression is typically supervised [30–32]
- **Clustering:** generally unsupervised, the learning algorithm divides inputs into a number of groups based on predetermined similarity measurements [30–32]
- **Dimensionality reduction:** involves reducing the number of random variables to which a set of inputs map, providing a lower-dimension mapping of inputs to outputs; topical categorization is a prime example [30–32]

2.2.3 Classification Problems

Classification, with respect to ML, involves making a prediction about the category, or categories in the case of multi-label classification, to which a previously unobserved event belongs [33]. In ML, models learn to make predictions based on training data with inputs whose class(es) are known. Sentiment prediction [18] and spam filtering [19] are practical examples of classification at work. Each has a set of labels; for sentiment prediction, this can be a list of any number of desired conveyed emotions, while, for spam filtering, usually only **spam** and **not spam** are required for labeling.

An algorithm that performs classification is called a *classifier*. A classifier, of which there are many types, is generated using a specific ML model type, such as linear regression or support vector machine, that learns associations between supplied training observations and their corresponding labels.

Support Vector Machines

A *support vector machine* (SVM) is a specific type of classifier that involves placing inputs into a high-dimensional feature space F of dimension D. Each input sample is placed as a data point in F, identified by a vector in F. An SVM algorithm generates a set of one or more hyperplanes (H) that separate the inputs into classes. A hyperplane x where $x \in H$ is a subspace within F with dimension $D - 1$ [34, 35].

SVMs are extremely useful for inputs that cannot be *linearly separated* in a two-dimensional space, i.e. it is not possible to construct a straight line boundary between data of different classes. When linear separation is not possible, a non-linear function ϕ projects the inputs into F, allowing for one or more hyperplanes to split the data points [36]. We will see more of how this works further in this section.

SVMs have found use in such applications as text categorization [37], mapping subcellular protein locations [38], image search engine optimization [20], and financial forecasting [39], to name a few.

For USBeSafe, we consider the use of the Schölkopf, *et al*. [40] one-class SVM (OCSVM) for the function of novelty detection. The OCSVM relies on training data coming from a single class and determines whether new inputs belong to that identified class or if they are novel observations. Successful novelty detection using OCSVMs depends on a clean training set;

there should be no outliers polluting the data that make generating a soft boundary around the observations as a hyperplane difficult [32].

OCSVM Specifications and Parameters

Like all ML algorithms, the OCSVM has specific and distinct goals during its training and testing phases. During OCSVM training, the overall goal is to find a function f which yields a positive (+) result when applied to a point within F and a negative (-) result otherwise [41]:

$$f(x) = \begin{cases} +1\,, \text{ if } x \in F \\ -1\,, \text{ if } x \notin F \end{cases}$$

During testing of new inputs, the output sign of f determines whether an input is belongs to the learned class or not.

To define the OCSVM, we first consider the inputs, or training observations [40]:

$$x_1, ..., x_n \in X$$

where $n \in N$ equals the number of training observations provided. Every observation in X must originate from a single class.

ϕ is the feature mapping $X \to F$ that represents all $x_i \in X, 1 \leq i \leq n$ lifted into the high dimensional space F. This mapping into F is achieved by the use of a *kernel*, generalized here:

$$K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j) \text{ where } i, j \in N \text{ [36]}$$

Four specific kernel functions are often applied to OCSVMs:

- **linear:** $K(x_i, x_j) = x_i x_j$ [42]
- **polynomial:** $K(x_i, x_j) = (\gamma x_i x_j + r)^d, \gamma > 0$ [42]
- **radial basis function (RBF):** $K(x_i, x_j) = e^{-\gamma ||x_i - x_j||^2}, \gamma > 0$ [42]
- **sigmoid:** $K(x_i, x_j) = tanh(\gamma x_i x_j + r)$ [43]

where $i, j \in N$. Parameters γ and r serve as mapping threshold coefficients that define boundary characteristics and the influence of individual inputs on how H is defined.

The OCSVM operates via an algorithm that returns a function f that yields +1 in the area (normal) defined by the inputs and -1 outside of this region (novel). To accomplish this, the Schölkopf, *et al*. OCSVM uses a method of separating each input in X from the origin using the following minimization function [40]:

$$\begin{aligned} \min_{w \in F, \xi \in N, \rho \in X} \quad & \tfrac{1}{2}||w||^2 + \tfrac{1}{\nu n} \textstyle\sum_{i=1}^{n} \xi_i - \rho \\ \text{subject to} \quad & w \cdot \phi(x_i) \geq \rho - \xi_i, \xi_i \geq 0. \end{aligned} \tag{2.1}$$

The parameter ν, where $0 < \nu \leq 1$, controls the tradeoff between two major goals: (1) maximizing the number of observations that return a +1 and (2) minimizing the *support vectors* used to determine the result [40].

Directly affecting one another, ν defines an upper bound for the fraction of training errors and a lower bound for the number of support vectors [32]. Support vectors are the data points that support, or lie closest to, H. Equation 2.1 is solved for w and ρ. The slack variables ξ_i allow for some flexibility in the case of erroneous points, such that not all points have to fall on the positive side of H [41].

Using the dual problem of the minimization in Equation 2.1, it is possible to solve for the coefficients (α) in Equation 2.2 [40]:

$$\begin{aligned} \min_{\alpha_{i\ldots n}} \quad & \tfrac{1}{2}\textstyle\sum_{i,j=1}^{n} \alpha_i \alpha_j k(x_i, x_j) \\ \text{subject to} \quad & 0 \leq \alpha_i \leq \tfrac{1}{\nu n}, \textstyle\sum_{i=1}^{n} \alpha_i = 0. \end{aligned} \tag{2.2}$$

These coefficients can then be applied to the decision function, Equation 2.3 derived in [40]:

$$f(x) = sgn(\sum_{i=1}^{n} \alpha_i k(x_i, x) - \rho) \tag{2.3}$$

An OCSVM that has learned its decision function (Equation 2.3) from a training set can apply the function to new input, predicting its class with a simple result of +1, belonging to the class the model on which the was trained, or -1, not belonging to the trained class.

2.2.4 Model Fitting

For an ML model to prove effective, the model needs to fit the data properly. This fitting can prove difficult to achieve, so model scoring relies on a number of measurements, to include *precision*, *recall*, and *accuracy*.

There exists the possibility of a model learning characteristics of the training data too specifically. *Overfitting* occurs when, instead of learning the overarching relationships among all observations, the model precisely describes noise and random errors [17, 32]. The model will actually *memorize* the training data instead of learn trends, leading to poor prediction performance for new observations previously unseen by the model.

In the case of novelty detection or binary classification, overfitting a model can lead to a high rate of type I errors, or false positives, of new data. A false positive (FP) occurs when the model flags the new observation as novel when the event actually is normal. Avoiding overfitting involves minimizing type I errors and maximizing the true negative rate, or the precision score, correctly identifying a normal observation as such [17].

Underfitting a model can also occur when the boundaries separating classes are too loosely defined. Underfitting is characterized by a high rate of type II errors, or false negatives, when examining new observations [32]. A false negative occurs when the model identifies a new observation as normal when it should be identified as novel. To correct underfitting, type II errors should be minimized and the true positive rate, or the recall score, should be maximized, correctly identifying a novel observation as such [17].

Table 2.4 illustrates a matrix of these result and error types according to truth and predictions.

TABLE 2.4: Overview of error and result types for novelty detection and binary classification.

		Truth	
		Novel	*Normal*
Prediction	*Novel*	true positive	false positive, type I error
	Normal	false negative, type II error	true negative

Parameter Estimation

Every type of ML model tends to have a number of parameters that can be set to help optimize performance and fit, such as those discussed in section 2.2.3 for OCSVMs. Often, researchers do not inherently know what parameter settings are most effective for the given model type and data. A *grid search* can prove useful in this situation, where possible values for each parameter are defined and models learn the input data according to every parameter combination [32].

Consider a model that has settings for three parameters, a, b, and c:

- a can take any value in $[a_1, a_2, a_3]$
- b can take any value in $[b_1, b_2]$
- c can take any value in $[c_1, c_2, c_3, c_4]$

In this case, the grid search algorithm would create 24 different parameter set options, using each combination of the three parameter types to generate a model instance that can be individually scored for performance.

k-Fold Cross Validation

Ultimately, an ML model needs to perform well when making predictions regarding new observations on which the model was not trained. To determine the efficacy of a specific model, we can use *k-fold cross validation*, or k-fold CV. This technique is used to divide input observations for model learning into separate training and testing sets [44]. $k \in \mathbb{N} \geq$ number of inputs and determines how many times to divide the corpus; for example, if $k = 5$, the corpus is split into five parts, or folds.

To determine model usefulness, the folds not used in each training session are used as the testing set as the previously unseen observations. In other words, each iteration of the 5-fold CV would train on 80% of the data and test on the other 20%, shown in Figure 2.2. Each model is scored according to its *accuracy*, a percentage of all test observations that are correctly classified based on known labels [32].

Grid searches combined with k-fold CV can be used for finding suitable model parameters. After such experiments occur, a new model is trained on the entire data set with the determined ideal parameters.

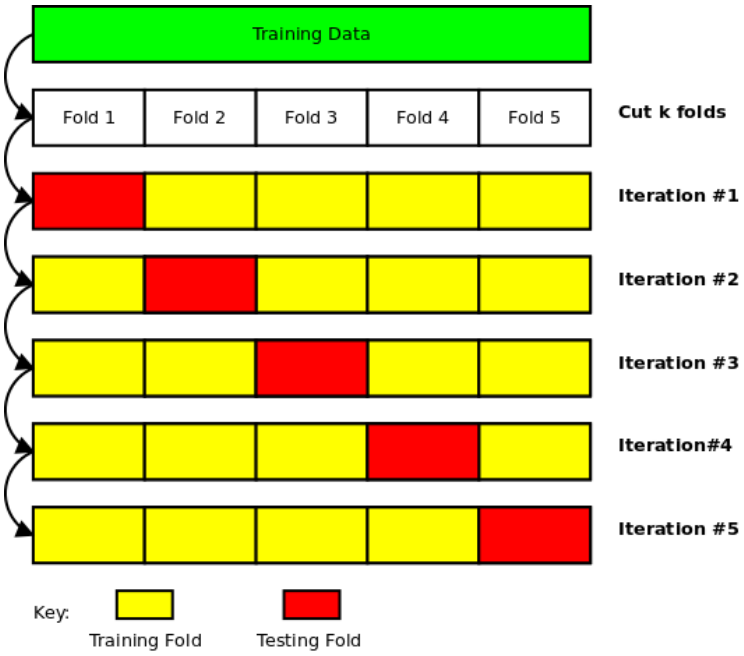

FIGURE 2.2: An example of k-fold CV, where $k = 5$.

2.3 Detecting Malicious Activity

In computing, an action is either benign or malicious. In essence, determining whether an action falls into one category or the other makes detection of malicious activity a classification problem. There are two general approaches to such a problem: (1) signature-based and (2) anomaly-based detection mechanisms. Signature-based detectors use massive databases of attack signatures (for example, the hash of a piece of malcode) and look for signature matches between a database of known malicious signatures and what it observes on a system [45, 46]. Practically speaking, signature-based systems are difficult to maintain because of the ease with which malware authors can change a piece of software while preserving its operational intent [47]. Adding, deleting, or modifying even a single character of a file will completely change the its signature.

2.3.1 Anomaly Detection

The other approach opposite of signature analysis, anomaly-based detection mechanisms rely on some notion of typical or standard activity as a baseline and raise an alert if any given activity does not fall within this realm [46]. While anomaly detectors are prone to false positives where an anomaly does not occur yet the detector raises an alert, discussed in 2.2.4, they offer more promise for detecting malicious activity because they do not rely on easily bypassed hard signature checks.

Because of USB traffic characteristics and means of transport, there exist strong parallels between literature regarding network anomaly detection (NAD) and work on USBeSafe. Like a network interface card (NIC), a USB

bus supports two-way, stateful packet-based communication that must adhere to a specific transport protocol. These similarities create a heavy reliance on the extensive previous work in NAD research.

The idea of flagging anomalous activity was first introduced by Anderson [48]. Anderson established means of characterizing normal computer usage according to types of processes, resource consumption rates, time parameters, file access, and device functions, all across a group of users. The proposed monitoring system would compile all of this information in real time and compare it against the known baseline in the *surveillance program*, the anomaly detector [48].

Eventually, Lane and Brodley [49] brought about the practice of applying ML techniques to anomaly detection. Using a simple binary classifier with a basic threshold metric, [49] was able to characterize user data of tokenized UNIX shell commands, achieving up to a high 99.2% detection accuracy score.

Both [49] and Bhuyan, *et al*. [50] provide extensive reviews of current NAD methods and their applications of ML, to include:

- ***Statistical systems*** apply inference tests to decide whether a new observation could be generated from the model. Like HIDE [51], these systems often rely on artificial neural networks (ANNs) that, while similar to Markov chain modeling [24], are more highly adaptive and base decisions on aggregated information at the end of the ANN system [52–54].

- ***Clustering***, typically used in data mining, can group data points in an offline environment according to some similarity measure and identify the outliers that do not seem to fit into any cluster. Clustering techniques have been utilized in systems such as [55, 56].

- ***Knowledge-based systems***, also called rule-based or expert systems, rely on *rule engines* that match defined rules against the current system state based on specific parameters. *Snort* [57], a classic open-source rule-based NAD system, applies rule-matching to each packet, looking for certain values in headers and payloads, and currently has over 20,000 active rules [50].

- ***Classification systems***, as discussed in section 2.2.3, applied to NAD categorize network traffic patterns into a number of classes, relying on training data with labels of class identification for classifier generation [41, 58, 59].

Other ML-based options have also proven effective, such as *Anagram* [60] and the Kruegel, *et al*. multi-model approach [61]. *Anagram* uses Bloom filter-based n-gram analysis to efficiently detect anomalous network payloads; each observation hashmaps to an element in a bit array, flipping the bit and generating a payload fingerprint. [61] uses a vastly different approach. Instead of a specific model-based system, Kruegel, *et al*. use a feature-based method: apply numerous models according to the feature sets extracted from network traffic, and the aggregate application of these models against test data determines whether network events are flagged as anomalous.

SVM-based NAD Systems

As SVMs are good at making generalizations about data, they have naturally made their way into NAD implementations with favorable levels of success. Often viewed as a classification-based problem, NAD research has successfully applied SVMs in a number of systems.

Mukkamala, *et al*. [58], in a comparison of SVM versus neural network performance, used SVM technology to create a binary classification NAD system for attack detection, using 41 different features from network traces obtained from DARPA. This system achieved a 99.5% accuracy rate using a two-class SVM, distinguishing between normal and attack behavior, regardless of attack type. In another SVM-based binary classifier system, Palmieri, *et al*. [59] achieved 97.7% accuracy and a 3.2% false positive rate when detecting whether card-sharing, a way for non-paying users to access digital television content on the Internet, was occurring on a network.

OCSVM implementations have also proven effective in network-based anomaly detection. Wang, *et al*. [62] improved upon the existing STIDE NAD by implementing a custom kernel and applying it to an OCSVM. This kernel decreased the false positive rate of the STIDE NAD from 13.4% to 4.5%, a marked improvement demonstrating the powerful capabilities of fine-tuned SVMs. Wagner, *et al*. [41] utilized a custom-kernel OCSVM to model Netflow records, concise representations of network traffic. [41] used an existing data set, assuming it clean of outliers, i.e. malicious Netflow records, as required for novelty detection, and tested the OCSVM against Netflow records containing 8 different attack types. False positive rates for each attack type were 3.3% or below [41].

2.4 Review of USB-based Attack Vectors

In this section, we briefly cover two well-known avenues of attack for a malicious actor to breach security measures using USB devices. These attack vectors take advantage of human weakness as well as convenience features built into the Windows OS.

2.4.1 Oops... I Dropped It

Regardless of how malware is stored on a device, a small amount of social engineering can lead to a case of "curiosity killed the cat". One of the most high-profile cyber attacks, Stuxnet, made its way into the Natanz nuclear facility simply because an employee found a USB flash drive laying in the parking lot and decided to view its contents by plugging it in to a facility system [3, 4].

In 2011, the U.S. Department of Homeland Security (DHS) performed a test where staff randomly dropped CD-ROMs and USB drives in government and contractor parking lots. The results show just how easy it can be to take advantage of this form of attack vector. 60% of those people who picked up one of these DHS-modified CDs or drives plugged it in to office machines out of curiosity, and that number jumped to 90% when the CD or USB drive had an official logo printed on it [63].

However amateur it may seem, time and time again, this form of attack vector works. Even after the 2008 DoD scare when all removable media

was banned [2], we see that humans are not infallible, and using nearly effortless social engineering to infiltrate target systems proves effective.

2.4.2 Autorun.inf

Autorun, built into Windows, was initially meant as a feature to the user, automatically performing specified sets of actions to prepare a CD-ROM for seamless use when mounted to a host [64]. A CD-ROM that utilizes this feature must have an *autorun.inf* file stored within its data section. When it is physically mounted to the host, Windows searches for the file and, if found, runs it. With the introduction of USB devices came U3 [65], a way for USB drives to emulate a CD-ROM partition in such a way that forced the hand of the Windows operating system to search for an *autorun.inf* file [46].

Until Windows Vista, there was no option to disable the Autorun feature; if the CD-ROM or USB drive possessed an *autorun.inf*, it would run the commands contained in the file, regardless of user desire to do so[64]. A disc or device with a malicious *autorun.inf* would certainly be hazardous. USB Switchblade, a tool developed to help automate malicious *autorun.inf* creation made exploitation of this attack vector trivial [66]. While software protections are now in place to block this attack vector, it ultimately fueled the motivation for future attack classes via USB.

2.5 BadUSB – A Novel Type of Attack

As mentioned earlier, a host machine is ultimately in charge of initiating all transactions with a USB device. The host is naive to the information that a device is going to send across the bus. This naivety is most obvious when the device reports it capabilities with various descriptors; the device can report any capabilities it has available. BadUSB leverages this fact to introduce a rather lethal class of attacks via USB hardware.

BadUSB [7, 8], introduced at BlackHat 2014 by Nohl and Lell, illustrates the unavoidable dangers of TD firmware: (1) it must run for the device to operate, otherwise a user could not complete any desired I/O functions, and (2) generally, firmware is not write-protected after a device boots [8]. In tandem, these two properties open up TDs and, in turn, the hosts with which they connect to a wide array of attacks. Nohl and Lell demonstrated a select few of these potential attacks [67]:

1. **Keyboard emulation:** Upon inserting a mass storage device, specifically a USB flash drive (with capabilities for Windows and Linux), into a host USB port, the device presents itself both as mass storage and a keyboard, covertly performs keyboard actions to open a command prompt, and downloads malware from a remote location for reflashing future USB TD firmware.

2. **Network card spoof:** After plugging in an Android smartphone to a victim host under the premise of charging the device, the Android, unknown to the user, presents itself as a NIC. DHCP overrides the default gateway over USB-Ethernet, and the host sends all network traffic through the Android device. Upon the user visiting PayPal.com,

for example, the device routes the request to a malicious server and records login credentials, all the while the user believing he simply was not able to log in.

While BadUSB also demonstrated a boot-sector rootkit attack in which a seemingly benign TD hijacked the bootloader process of a host and planted a rootkit to gain persistence [67], the novelty of the first two attack examples make them far more interesting and dangerous. Rogue-TD attacks like those demonstrated in BadUSB, as we consider them, are unique and specific because they rely on both:

- Custom firmware with malicious intention residing on a USB device, and
- An unknowing user facilitating operation of a seemingly benign device on a host machine to allow the malicious firmware to run

By rewriting the firmware of an existing USB device, it is possible to hide malware in the code that communicates with a host. Malware scanners do not have insight into firmware code, which means a host cannot detect malicious firmware. Yet, that very firmware can communicate with and change host software as well as perform any number of other adversary-desired actions.

Consider a USB mass storage device as an example for how a rogue-TD attack can work. After the device is physically mounted to a USB port, a number of steps occur as discussed in 2.1.3. Figure 2.3 illustrates relevant communication that occurs between host and device as the device is enumerated. Of specific interest, when the host sends a Get Descriptor request to the device, the host knows nothing about device capabilities. This USB drive eventually reports that it possesses two interfaces, and the host requests that information from the device. To avoid suspicion by providing normal function to the user, the first interface reports mass storage capabilities.

The second interface, on the other hand, reports HID capabilities, unbeknownst to the user. The OS simply acts based on information provided by the device and will happily load a driver to accept the USB drive as a HID device. Code in the malicious firmware then can open a command prompt, for example, to perform privilege escalation, exfiltrate files, or copy itself for further device propagation.

2.5.1 Existing Defenses and Limitations

Defending against rogue-TD attacks is vital when it comes to overall system security. Because these attacks are difficult to detect and can be tailored for any intent, they have the ability to inflict large amounts of damage. Two potential solutions proposed by Duckling [68] unfortunately fall short in terms of an ideal protection mechanism:

1. Device manufacturers could hardwire USB microcontrollers to only allow firmware updates that are digitally signed by the manufacturer. The issue with this approach is that the device is effectively vendor-locked and can only be updated by the manufacturer, deeply compromising interoperability/usability. Additionally, this solution assumes that the manufacturer's private key is not compromised.

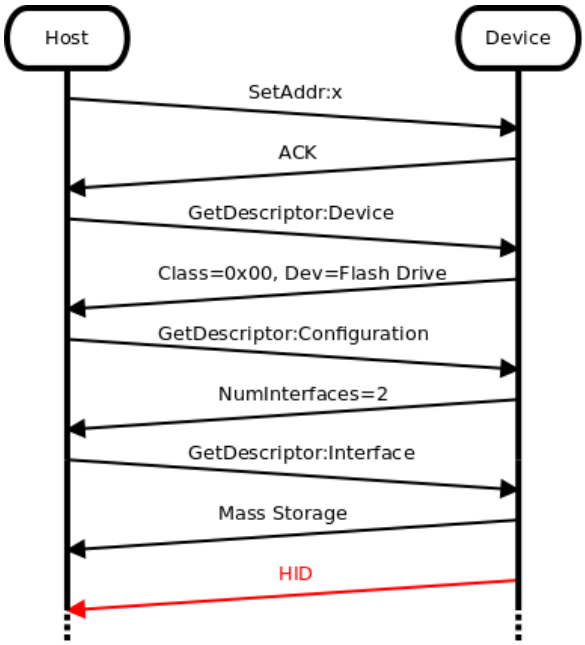

FIGURE 2.3: A malicious device sends a covert interface descriptor to the host with the user unaware of activity.

2. Manufacturers could incorporate hardware interlocks into devices, meaning that an end-user may have to press a button or toggle a switch on the device to allow a firmware update. A major concern with this solution is user education and awareness. It is entirely possible for the malware residing on the victim host in the keyboard emulation BadUSB attack to prompt the user for a switch toggle to "update" the firmware on a newly inserted device. Without appropriate knowledge of how to look for and examine potential threats, many users may blindly obey the request, negating the intention of a hardware interlock.

Both of these propositions rely on significant changes to USB device operation and shift tremendous burden on manufacturers and users to adjust how they create and use USB devices. In the remainder of this section, we explore existing solutions to the rogue-TD attack vector and their limitations.

IEEE 1667

Currently, the *de facto* technology available for protecting against malicious bytes residing on and executing from a device exists in IEEE Standard 1667, *Standard Protocol for Authentication in Host Attachments of Transient Storage Device* [69]. Implemented in Windows Vista SP2 OS and higher as Windows Enhanced Storage [70] and built into devices as custom firmware by manufacturers, IEEE 1667 seeks to create a means for bidirectional authentication via an X.509 certificate infrastructure between hosts and devices. Described as typically being used for devices with storage partitions, key

exchanges and certificate checks are performed prior to device storage access as a way of providing protection against unknown, and potentially malicious, devices and/or hosts [69].

The fundamental premise is that if a device's firmware can decide the presumably safe hosts with which it successfully pairs, it is much less likely that malware could migrate onto the device, unknown to the end-user. Due to the standard's bidirectional nature, this premise can also be applied in the opposite direction with the host deciding the devices with which it will pair.

Unfortunately, adoption of IEEE 1667 has been slow since the standard was first introduced in 2006, as only a handful of devices leave the manufacturer provisioned as 1667-compliant [71–74], and none of these are even USB devices. Consequently, 0% of USB devices are provisioned to possess any sort of entity authentication mechanism as a means of vouching for the safety of data residing on the device; we must ignore devices such as IronKey, which exist for the purpose of confidentiality protection [75]. Furthermore, it follows that organizations allowing TDs in any capacity are unlikely to check for 1667-compliant devices upon pairing of device to host. Such a situation dramatically increases the surface area for attacks, a prominent reason why rogue-TD attacks can possess such potency and effectiveness.

Because of the broad nature of rogue-TD attacks and the extremely small likelihood of redefining the USB specification to mitigate the issue, developing defensive techniques lies in the hands of those that have a greater vested interest in systems security. Without changing the fundamentals of USB communication, patches for all major operating systems need to be in place to minimize the attack surface for any adversary.

Linux and the GoodUSB Response

GoodUSB [76], a proposed fix to issues brought forth by BadUSB, shifts the burden of responsibility to the user when it comes to security. Implemented in the Linux kernel, GoodUSB mediates enumeration interactions between the host and the device by: (1) involving the user, asking for recognition and verification input, and (2) matching user-expectation policies to what the device claims as its functionality, monitoring device activity if viewed as suspicious.

GoodUSB is a three-pronged kernel module utilizing a *USB mediator* that performs the following tasks [76]:

- **Policy creation and enforcement:** authorizes USB actions according to user expectations encoded into policies.
- **Device recognition:** record device enumerations to allow the OS to recognize whether it has already seen a given device.
- **Honeypot analysis**: suspicious activity is profiled in a virtual honeypot to provide insight to the user into device behavior.

One of the major enhancements to the Linux kernel by GoodUSB is the ability to suspend loading of a device driver until it confirms policy-matched functionality. Figure 2.4 [76] illustrates this feature as well as the

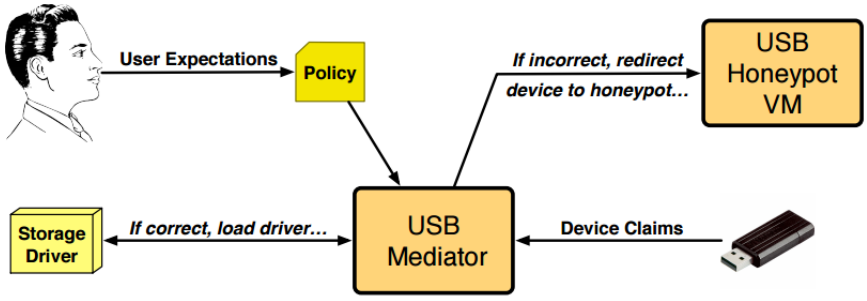

FIGURE 2.4: GoodUSB infrastructure centers around a USB mediator that enforces policies and monitors USB activity [76].

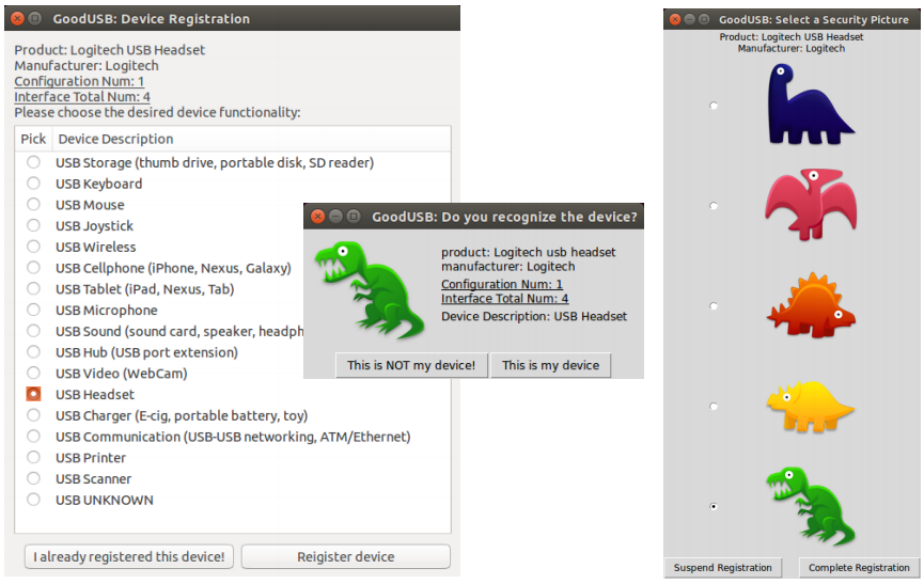

FIGURE 2.5: GoodUSB requires a user to specify device functionality each time a new device is introduced to a new host for policy generation [76].

main infrastructure of GoodUSB. It is implied that the user introduces a device to the system. The user then generates a policy according to what he expects with regards to device functionality, as seen in Figure 2.5 [76]. The mediator compares that policy, whether new or matched to a previously seen device, to what the device claims it can do. If the two match, the mediator loads the proper device driver; otherwise, device activity is redirected to a virtual honeypot, where it is monitored for potentially malicious behavior.

Unfortunately, GoodUSB relies heavily on user interaction, forcing users to identify devices being plugged into a host as well as indicate their desired functionality. Even though the mediator's performance overhead is minor at about 5% [76], the problem with GoodUSB lies in its usability. With the number and transportability of USB devices in end-user computing, effective use of GoodUSB would require $x * y$ *first enumerations* by the user to

create policies for the devices, where x is the number of USB devices and y is the number of host machines, a significant and tedious setup overhead.

Windows Patch 3143142

An extensive Windows patch (Security Update 3143142) for all supported operating systems released in March 2016 sought to close the software hole that enables privilege escalation by BadUSB-style attacks. Windows described the vulnerability as a driver weakness in which the Windows USB Mass Storage Class driver could not correctly validate certain memory objects [77]. As of this writing, there has been no published patch effectiveness analysis.

2.5.2 Enter: USBeSafe

In response to these existing solutions to oue-TD attacks discussed in the previous section, we recognize that a more ideal solution incorporates a system relying on as little change as possible to the end-user operational *status quo*. This system should:

- Require little to no end-user interaction
- Avoid manufacturer-based hardware changes
- Incorporate the protection mechanism into the operating system

To meet these goals, we introduce USBeSafe, a first-of-it-kind application of OCSVM to USB traffic, providing an infrastructure for future offline and live USB event anomaly detection systems. USBeSafe leverages the powerful generalization capabilities of OCSVM to learn various feature patterns specific to USB traffic. While USBeSafe has an extensible design incorporated into each phase to add device classes, features, model types, etc., we focus specifically on USB keyboard traffic and the features that characterize such traffic as benign. This focus stems from a goal of detecting a specific type of rogue-TD attack: determining whether a covert HID configuration is present and active on a device, sending input across the bus to the host machine, as shown in Figure 2.3.

Chapter 3

Formalization and Implementation

In this chapter, we discuss the USBeSafe operational framework in detail. First, we define the threat model upon which USBeSafe was built. Next, we provide a brief overview of the six components that define execution and data flow in USBeSafe. Then, each subprocess is discussed in depth, paying careful attention to inputs and resulting outputs.

3.1 Threat Modeling

Because of the wide scope of possible malware that can run with access to a machine, it is valuable to define a threat model that is adversary-centric. For USBeSafe, we frame our adversary according to his capabilities and the attack environment:

- He can reflash the firmware of a USB device to emulate a USB keyboard.
- He has means of pairing the malicious device with a target machine, though he does not require physical access to the machine.
- He has no insight into the keystroke patterns of the target machine's user.
- The malicious device must enumerate via USB with a vulnerable target machine for the attack to run.
- There exists no USB level authentication mechanism between the device and the target host.
- The target machine's operating system has known states for when keyboard input will result in a visible action. For example, when a command prompt is open, it is possible to send keystrokes that result in text on the terminal window.
- The malicious firmware can monitor for or force the target machine's operating system into a state ready for keyboard input.

USBeSafe aims to serve as a safety mechanism for an adversary scoped by all of the above by detecting anomalous USB HID traffic.

3.2 System Overview

USBeSafe is a USB event anomaly detector that uses a corpus of known benign USB activity to train ML models in making predictions about whether new, previously unseen USB activity is potentially malicious. Because of the connection between our work and NAD, there are strong parallels regarding system infrastructure and operation. [50] outlines the typical execution order of both live and offline NAD systems, forming the basis for our approach:

1. Traffic capture
2. Data preprocessing
3. Application of matching mechanism(s)
4. Taking actions based on results of step **3**

The USBeSafe infrastructure takes on a modified form of [50], shown in Figure 3.1, providing a more structured and specific approach to the anomaly detection framework. USBeSafe is implemented in Python, utilizing the `scikit-learn` [32] library for ML-based development.

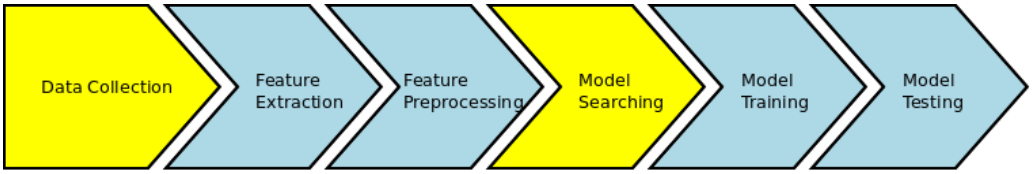

FIGURE 3.1: USBeSafe infrastructure and system flow; processes in yellow can be continuously repeated for system refinement.

In the remaining sections of this chapter, we explore each process in Figure 3.1 in depth, with focus on procedures, data flow, and resulting products.

3.3 Data Collection

3.3.1 `usbmon`

For any ML-based system to prove effective, training data needs to be collected for models to learn, generating a baseline for future anomaly detection. For data collection in the development of USBeSafe, we utilize the Linux kernel module `usbmon`, located at `/sys/kernel/debug/usb/`, to capture I/O traffic moving across a monitored USB bus. `usbmon` works by capturing and passing observed USB packets to the Host Controller Drivers in Linux, much like `tcpdump` can be used to monitor network traffic [78].

To generate a meaningful training data corpus, we rely on "normal" usage of these devices. Though it is difficult to characterize what normal usage looks like from an end-user action-based standpoint, we combat this issue by simple persistent monitoring of USB events with `usbmon` on a host

machine. Instead of developing a series of specific actions to perform, persistent monitoring beginning at boot time allows for USB event data that stems directly from everyday computer usage. In turn, more realistic characterizations of the data can be made further in the execution flow of USBeSafe.

3.3.2 Traffic Capture

We set up `usbmon` on a Linux Ubuntu 14.04 LTS 64-bit host machine, segregating traffic monitoring to a single USB bus with physical access to two USB ports on the outside of the machine. Each time `usbmon` starts on boot, it generates a new `PCAP` file, appending live traffic as the module observes new packets across the monitored USB bus. On system shutdown, `usbmon` saves the generated `PCAP` to disk. We define a *trace file* to be the `PCAP` file generated over the duration from host machine boot to shutdown.

3.3.3 Understanding the Data

Each time `usbmon` captures USB traffic over the lifecycle of a host, it creates a `PCAP` file containing each packet in sequence observed on the USB bus it monitors. Wireshark [79], a popular and useful `PCAP` file parser, contains built-in capability to parse and interpret USB packets from a trace file.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	host	2.0	USB	64	GET DESCRIPTOR Request DEVICE
2	0.004347000	2.0	host	USB	82	GET DESCRIPTOR Response DEVICE
3	0.004370000	host	1.0	USB	64	GET DESCRIPTOR Request DEVICE
4	0.004375000	1.0	host	USB	82	GET DESCRIPTOR Response DEVICE
5	181.156984000	2.1	host	USB	72	URB_INTERRUPT in
6	181.157101000	host	2.1	USB	64	URB_INTERRUPT in
7	181.213006000	2.1	host	USB	72	URB_INTERRUPT in
8	181.213102000	host	2.1	USB	64	URB_INTERRUPT in
9	181.277038000	2.1	host	USB	72	URB_INTERRUPT in
10	181.277135000	host	2.1	USB	64	URB_INTERRUPT in
11	181.325054000	2.1	host	USB	72	URB_INTERRUPT in
12	181.325151000	host	2.1	USB	64	URB_INTERRUPT in
13	181.437121000	2.1	host	USB	72	URB_INTERRUPT in
14	181.437229000	host	2.1	USB	64	URB_INTERRUPT in
15	181.509148000	2.1	host	USB	72	URB_INTERRUPT in

FIGURE 3.2: Sample Wireshark representation of a `usbmon` trace.

Figure 3.2 shows the enumeration and some post-configuration traffic for a keyboard. First, the host requests device descriptor information from device 2.0, corresponding to bus 2 and default device address 0, yielding source 2.0. The device responds with some preliminary information, and the host requests the full device descriptor. After the host receives the descriptor, it assigns a permanent address to the device (0x01), loads a driver, and steady-state traffic begins. Section 2.1.3 contains detailed information on this process. We recognize that this behavior does not directly match the expected descriptor hierarchy presented in Figure 2.1 and discuss this issue further in section 4.1.1.

For each keystroke, two packets are transferred. First, the device sends a `URB_INTERRUPT` packet to the host; upon receiving the interrupt containing the payload that identifies the key pressed, the host sends back an ACK.

Here, a *URB*, or USB request block, is simply the wrapper used in Linux to package a USB packet, containing all USB header information. This `URB_INTERRUPT` is analogous to the interrupt transfer discussed in 2.1.3. Drawing one-to-one relationships to all USB transfer types, the other Linux equivalents are `URB_CONTROL`, `URB_BULK`, and `URB_ISOCHRONOUS`. For any packet that contains a payload, such as the interrupt transfers shown in Figure 3.1, the last two fields of the packet contain the payload length and the payload itself.

3.4 Feature Extraction

To train an ML model, we must extract features from the corpus that effectively characterize the data. When dealing with a large corpus containing complex data and communication structures like USB, this extraction phase is most clearly defined as a two step process: (1) data preprocessing, and (2) pulling out features from the preprocessed data. We discuss at length how each of these steps are implemented in the USBeSafe system.

3.4.1 Data Preprocessing

Feature extraction involves extensive work in data preprocessing; for USBeSafe, this means molding the supplied corpus in such a way that the data takes on order and meaning. In our system, the feature extraction phase allows us to supply any number of `PCAP` files, with the precondition that they must have been previously generated by `usbmon`. We take the following steps to preprocess any supplied `PCAP` files:

1. **Packet-based PCAP parsing:** Using the `pcapy` [80] Python library, for each `PCAP` supplied, parse each raw USB packet within and form it into a `TraceEvent` containing the URB and USB header information and any payload. Each `TraceEvent`, representing a single USB packet, is added to a `Trace`, a list of `TraceEvents`, one for each `PCAP` trace file. The `TraceLibrary` contains all `Traces`.

2. **Extract (Bus ID, Device ID) subtraces:** Each USB bus can support interactions with up to 127 devices simultaneously. To identify a physical connection between a host and device, we generate a tuple from each `TraceEvent`, containing the host bus ID as well as the assigned device ID on the bus, ranging [0, 127]. For each trace in the `TraceLibrary`, we sort existing `TraceEvents` into new `Traces` contained in `TraceLists`, with each `TraceList`having an associated (bus ID, device ID) tuple. This action reforms the `TraceLibrary` into a set of `TraceLists`.

3. **Chronological packet sort:** Each `TraceList` in the `TraceLibrary` contains a number of `Traces`. Each `Trace` contains a series of `TraceEvents` that have fields for time since `usbmon` started the monitor, shown in Figure 3.2 The `TraceEvents` in each `Trace` are sorted according to timestamp, from earliest to most recent.

4. **Enumeration cycle breakdown:** Each time a host begins the enumeration process with a USB device, we see a Get Descriptor request/response pair for the device. Our goal here is for each `Trace` to represent an individual lifecycle of a device, from enumeration, to configured communication, to termination. For this reason, we further break down each `Trace`, parsing out a different `Trace` each time we encounter this event pair in two consecutive `TraceEvents`, ending any previous `Trace`.

5. **Load descriptors:** For each `Trace`, we identify the device and configuration descriptor responses, storing them as auxillary information for the `Trace`. Each `Trace` contains a `DeviceDescriptor` and `ConfigDescriptor` if the corresponding packets are found in the trace. `usbmon` packages all interface descriptor information with the corresponding configuration descriptor, so we store any interface information with the respective `ConfigDescriptor`.

6. **Device class sort:** Device classes help the host identify device type and expected functionality; both device and interface descriptors contain class codes, according to the USB protocol. A `Trace` will contain one or more device class codes based on the extracted information in the `DeviceDescriptor` and any available `ConfigDescriptor(s)`. Class codes are defined on two levels, as shown in Table 2.3, and we sort each `Trace` into class buckets as specifically as possible, where deeper in the descriptor hierarchy is more specific. Any `Trace` with a class code not of interest is thrown out.

It is useful to understand the data structure hierarchy generated during data preprocessing. Figure 3.3 illustrates this hierarchy through loading the descriptor information, step 5. We lose this structure after the final step (6), when we sort useful `Traces` into buckets according to USB class code.

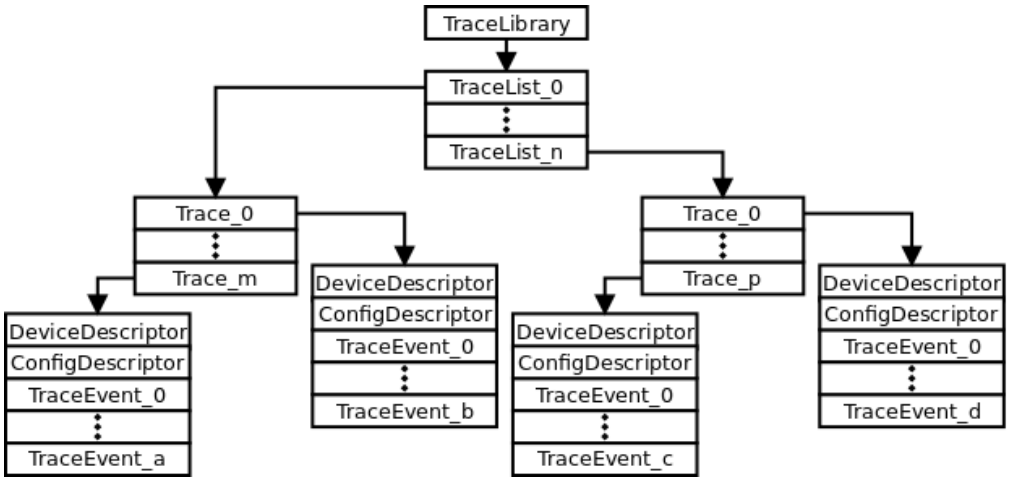

FIGURE 3.3: The data structure hierarchy after loading all device and configuration descriptors, through step 5 in section 3.4.1.

3.4.2 Potential Feature Selection and Extraction

To extract features from the data, we first need to determine what potential features we would like to use when training the USBeSafe ML model. We make the distinction of *potential* features here because the model search, further down the USBeSafe pipeline in section 3.6, provides us with feedback on what features, along with other factors discussed later, provide the best model accuracy. In selecting possible features, we look at avenues for pulling traits from the USB traces that are likely to define the data in ways that can characterize our data as benign. Regardless of the ML mechanism employed, feature extraction needs to provide a sound representation of the data collected.

With NAD ML systems, the types of features extracted typically vary widely based on what the researchers deem important and potentially useful [24, 50, 52–56, 58–61, 81, 82]. Because potential features in our research arena are likely partially dependent on device class, it is important to remember that, at a high level, USBeSafe aims to characterize benign USB keyboard traffic; this fact should inform selection of potential features.

Packet Interarrival Times

For characterizing USB keyboard traffic across a bus, one obvious route is to examine timing information for packets. Timing data can help reveal potential patterns in the ways a user types or how the bus manages USB traffic.

As one possible feature, we use the interarrival times, referenced in this paper as *itimes*, measured in milliseconds, between one packet and the next for all `TraceEvents`. Every first packet in a `Trace` has an itime of 0ms because no packet is transmitted before the initial packet.

It is not practical to use raw itimes as a feature because there is no upper bound for a given itime; a user may perform a keystroke after a few seconds or many days. To combat this problem of potentially infinite itimes in certain situations, we explicitly define an upper bound. To do this, we first define the following terms, noting the term interdependency:

- ***pause*:** the maximum time allowed to elapse before a new session begins
- ***session*:** a series of USB keyboard packets where each itime within the series does not exceed a specified pause length

Defining a `pause` value allows us to set the maximum itime between two `TraceEvents` before we consider the user to be starting a new typing session. To help determine an optimal `pause`, we examined three initial candidates (in milliseconds): 20000, 40000, and 60000. For each `pause`, we normalized the itimes for class codes 0x00 and 0x03; when a raw itime $i \geq$ `pause`, we set $i = 0$, thereby starting a new session.

Generating histograms for these itime collections based on USB class code, with varying `pause` values and interval lengths, we see some common patterns, found in Appendix B. An interesting observation for this HID keyboard traffic is that, regardless of `pause` length, localized modality occurs approximately every 4000ms, or 4 seconds, with large spikes in the

number of packets that transmit during these times relative to the start of a session.

To determine the impact a `pause` value has on the USBeSafe implementation, we performed an initial model search following the search strategy in section 3.6 to determine performance difference among different values with itimes as the sole feature. Ultimately, the results of this initial search revealed there was minimal performance difference among the `pause` values used, indicating to us that the value we choose is not consequential to overall model performance. For this reason, we set the `pause` to our first value of 20000ms.

Packet Type

A second feature type chosen as a viable candidate for future model generation are the values that define the type and purpose of each packet. The packet type is determined by two fields in each `TraceEvent`:

1. **URB type:** takes on one of two values to indicate whether there is an ongoing transaction, `URB_SUBMIT` (0x53), or if a transaction is complete, `URB_COMPLETE` (0x43).

2. **URB transfer type:** analogous to the four USB transfer types outlined in section 2.1.3, `URB_INTERRUPT` (0x01), `URB_CONTROL` (0x02), `URB_BULK` (0x03), and `URB_ISOCHRONOUS` (0x04).

Packet Payload

Finally, we deem it useful to partially characterize USB traffic, regardless of device class, by any payload contained within individual packets. Payload examination is a transparent means to determining patterns in data found within each `TraceEvent`. Similar to previous work [83, 84], we use a byte histogram to measure value frequencies, splitting the space of 256 values (from 0x00 to 0xFF) into 16 equal intervals, or bins.

Extraction and Storage

To extract the desired features, we generate a `FeatureListing` for each `TraceEvent`:

```
FeatureListing = [itime,
                  normalized_itime,
                  event_type,
                  transfer_type,
                  data_histogram]
```

During extraction, we store both raw itimes and any `pause`-normalized itimes; in this case, we store 20000ms-normalized itimes for each `TraceEvent`. These `FeatureListings` are stored, according to the hierarchy outlined in Figure 3.3 and sorted by device class, as a single JSON object, prepared for the next stage of the USBeSafe pipeline. We will call this file *features.extracted*.

3.5 Feature Preprocessing

After all features have been generated and stored in *features.extracted*, the next step is prepare them for input into the ML algorithm, when applicable, for model searching in section 3.6 and training in section 3.7. Any *.extracted* files supplied to this subprocess are combined, data that can be scaled is treated accordingly, and the features generated from each `TraceEvent` are molded into developer-defined n-grams and, ultimately, a list of *feature vectors*, where each vector contains all feature observations for a given feature across all `TraceEvents`. After feature preprocessing is complete, a single output file we will call *ngrams.preprocessed*, containing all feature vectors, is saved to disk.

3.5.1 Scaling

Many ML algorithms prefer to work with data that is scaled between two numbers, when possible, including USBeSafe's OCSVM implementation. For this reason, we scale values for feature types that have defined lower and upper bounds, specifically itimes and packet types.

Normalized itimes can be scaled because they, like any future test instances, are limited to a maximum value of 19999ms according to the predefined maximum `pause` of 20000ms. To decrease compute time down the pipeline by optimizing for OCSVM algorithms, we scale these normalized itimes in the range [0, 1].

The packet type is defined by two values in each packet, `event_type` and `transfer_type`. To store the packet type as a feature, we add the value of `event_type` and `transfer_type` for each packet, yielding eight distinct possible values. Furthermore, we scale each of these summations in the range [0, 1].

3.5.2 n-grams

Discussed in section 3.5.2, n-grams can help preserve information, diminishing loss that comes from training models based on single observations. For this reason, USBeSafe models can be trained on n-grams of varying n specified by the developer.

The OCSVM algorithm from [32], implemented in sections 3.6 and 3.7, requires a two-dimensional matrix of real numbers as input for both training and testing. Each row represents feature observations for a given packet in series, and each column represents a feature vector. To meet this matrix requirement, n-grams are generated according to the structure outlined in Table 3.1. An n-gram for a given packet p uses a sliding window that contains all `FeatureListings` in the range $[p, p + n)$. Packets near the end of the sequence simply have less information contained within their respective windows depending on the length of n. Of particular interest is the payload histogram feature type; each histogram observation must be broken into 16 values to meet the OCSVM algorithm input requirement.

The *ngrams.preprocessed* file generated as an output of this process contains n-grams for all possible feature types across all `FeatureListings` in *features.extracted*.

TABLE 3.1: The input matrix structure for the OCSVM algorithm dictates feature vector generation from feature observations for x packets with a sliding window of size n.

		Feature Vector												
		itime$_1$	...	**itime**$_n$	**ptype**$_1$	...	**ptype**$_n$	**hist[1]**$_1$	...	**hist[16]**$_1$	...	**hist[1]**$_n$	...	**hist[16]**$_n$
Packet	**1**													
	1													
	3													
	...													
	x													

3.6 Model Searching

After all n-grams are extracted from possible features and *ngrams.preprocessed* is generated, we perform a model search. The goal of a model search, for a given model type, is to pinpoint the best parameter values and feature types that result in the highest possible score assigned to a specific model instance.

USBeSafe's model search infrastructure utilizes an OCSVM grid search with internal k-fold CV. We use an OCSVM for two reasons:

1. The data collected in section 3.3 is from a single class, all considered to be benign (or normal).
2. The goal of USBeSafe is to detect novel observations, distinguishing from the one-class data on which the model is trained.

A thorough discussion of the results from the implemented model search performed in this research can be found in Chapter 4.2.3.

3.6.1 Framing the Search Space

To determine the search space, or the total number of models generated, we examine a number of different attributes upon which the model search relies:

- number of values of n for n-gram windows, a
- number of sets of modeled features for model input, b
- number of OCSVM parameter settings for model generation, c
- number of folds for k-fold CV, d
- number of scoring mechanisms used for model evaluation, e

Ultimately, the model search algorithm generates, trains, and evaluates $a * b * c * d * e$ models, taking all combinations of settings and inputs.

3.6.2 Search Algorithm

The model search algorithm operates as follows:

1. Extract feature vectors from *ngrams.preprocessed* according to the supplied n-gram length and requested set of feature types.

2. For every scoring mechanism, instantiate a grid search environment E, supplying the parameter space (with all possible defined OCSVM parameter settings), number of folds for k-fold CV, and the matrix of feature vectors.

   (a) For every parameter setting and fold combination in E, fit an OCSVM M to the training data and test M against the untrained fold using the scoring mechanism.

For each feature type set, n-gram length, and scoring mechanism, the output of this algorithm is a searchable grid, with an individual entry for each OCSVM parameter setting containing the following:

- parameter settings used to generate the model
- scores for each k-fold CV iteration
- average score across all folds
- standard deviation of the k-fold CV scores

3.7 Model Training

The model training subprocess of USBeSafe is fairly straightforward. After any desired OCSVM parameters have been found using the model search, we can use the model trainer to generate and persistently store one or more OCSVMs. As input, it accepts:

1. *ngrams.preprocessed*, generated in section 3.5
2. *requests.train* containing a list of tuples, one per line, each defining attribute settings to apply to an OCSVM trained on *ngrams.preprocessed*. Each tuple is constructed according to the following format:

   (USB class code, [feature types], {parameter settings}, n-gram size)

For reference, Appendix C lists the file contents of *requests.train* used for model training during experimentation in section 4.3. For each class code-specific model request m in *requests.train*, an OCSVM is generated with `sklearn.svm.OneClassSVM()` [32] according to the parameter settings in m. Each OCSVM is trained on the corresponding feature types in m from the supplied *ngrams.preprocessed* file. In this training, no k-fold CV occurs; the model learns the entire training set. Using [32], we are also able to achieve model persistence, storing each model, including its parameter settings, inside a class-based file hierarchy for future model loading. Finally, the algorithm creates an output text file, which we will call *models.trained*, containing a one-line tuple per trained model with the following information:

(USB class code, [feature types], {parameter settings}, n-gram size, model path)

The information found in *models.trained* is used during execution in section 3.8.

3.8 Model Testing

The final step of the USBeSafe pipeline involves applying one or more models trained in section 3.7 against one or more test trace files. These trace files contain USB bus traffic that the model has never before seen. We supply the test infrastructure with two inputs:

1. A file structured in the form of *models.trained* in section 3.7.
2. A file structured in the form of *ngrams.preprocessed*. The following preconditions apply:
   (a) The file must have been created using the same n-gram window size as the training data of the OCSVM against which it is being tested.
   (b) The file should contain the desired test vectors to test against the OCSVM and is expected to be data on which the OCSVM was not trained.

At this point, important distinctions are made among the terms *anomaly*, *novel*, and *malicious*. We consider *anomaly* and *novel* to be analogous terms, as they both indicate an observation is uncharacteristic of the model. An observation that is novel and an anomaly is **potentially** malicious. We do not classify an observation as malicious if it is classified by an OCSVM as novel (-1); rather, it is identified as possible malicious activity.

The model testing process loads file (2), containing the input observations on which a given model will make predictions, then for each request in *requests.train*, the tester will load the requested model, invoking `sklearn.svm.OneClassSVM.predict()` with the set of input observations. For each invocation of `predict()`, a list of prediction classifications is returned, with an entry for each input observation from the test trace. Each entry is one of {+1, -1}, where +1 means the OCSVM predicts the observation falls within the trained class, and -1 means the OCSVM predicts the observation falls outside the trained class and is therefore novel. We score the model's performance according to the following metric (also called the detection score):

$$\text{novel observation score} = \frac{\text{\# of input observations classified novel}}{\text{total \# of input observations}}$$

The output of the tester is a file where each line represents a test request containing the novel observation score for a given request and its associated model attributes.

Chapter 4

Experimentation and Results

In this chapter, we consider our experimentation setup and results in three distinct sections. First, we provide a basic overview of the trace data collected from `usbmon` and its implications for feature generation. Next, we discuss the model search phase at length; here, the bulk of our experiment took place, generating and evaluating a large number of OCSVM instances based on our declared search domain. Finally, we move to the model training and testing experiments; we use results from our model search to inform how we train OCSVM models before applying them against a known malicious trace file.

4.1 Data Collection and Feature Generation

For the development of USBeSafe, `usbmon` collected data over the course of 8 months, representing approximately two academic terms of USB device interaction. Over this span, 124 trace files were collected, consisting of 133.14 MB of raw trace file data. From this collection, we obtained a total of 1,235,094 USB event packets based on `usbmon` semantics. Though the actual number of USB packets that crossed the bus is greater, `usbmon` combines each series of (token packet, optional data packet, status packet) into one interaction. Any `usbmon`-based USB packet actually represents two or three USB packets on the bus, depending on whether the interaction included a payload or not.

Though we allow for extensibility in terms of monitoring a number of device classes, our primary detection focus is on USB traffic originating from a covert keyboard interface on a device. For this reason, we sort each `TraceEvent` in section 3.4.1 based on USB device class code. The HID device class, which keyboards fall under, is defined by class code 0x03, found in an interface descriptor.

4.1.1 Expectation vs. Reality

During data collection, preprocessing, and feature generation, we found that not all data conforms to expectations. Though all traffic meets the USB standard (communication errors would occur otherwise), some abnormal functionality was observed with regards to device enumeration.

Over the course of data collection, traffic from many keyboards was monitored, furthering the idea of "normal" usage. Not all keyboards reported a class code of 0x03. Ignoring host requests, expected keyboard enumeration involves:

1. Supplying a device descriptor with class code 0x00, meaning the host should use a code in the interface descriptor, and the number of configurations, ≥ 1.

2. For the configuration containing the keyboard interface, supplying the configuration descriptor with the number of interfaces belonging to the configuration.

3. For the keyboard interface, supplying an interface descriptor containing class code 0x03.

Instead, we found that some keyboards continue to function properly by communicating a device descriptor only, with no configurations defined, like that shown in Figure 3.2. These device descriptors contain USB class code 0x00, which indicates that a more specific class code can be found in the interface descriptor. Yet, there exists no interface descriptor. Though this occurs, each observed instance of this event sequence yields a successfully enumerated device, and the host accepts keyboard input immediately after receiving the device descriptor. At present, we are not sure to what to attribute this behavior. For this reason, we work around the issue during class bucketization in the feature extraction phase, moving all 0x00 traffic into the 0x03 bucket containing keyboard traffic identified as such in an interface descriptor.

Another issue we had to account for in loading the trace file data into usable structures was what action to take when encountering a malformed packet. In some instances when the host would request a device descriptor, the device would respond with a malformed descriptor packet, forcing the host to make the request again. As the purpose of the search for descriptor responses was to load information into the `TraceLibrary`, we chose to ignore these request/response pairs when they did occur.

4.2 Model Searching

4.2.1 Performing the Search

Discussed in section 3.6, the size of the search space depends on a number of attributes, each discussed at length below. A summary of these attributes is shown in Table 4.1, calculating a total of 5,880 OCSVM model instances generated during experimentation, covering all combinations of possible attributes.

n-gram Possibilities

Choosing possible values for n is a rather arbitrary process; for this reason, we use two values for n: 1, 2. Unigrams (or 1-grams) are used as a simple baseline in this experiment. Because of the knowledge loss with such a low n, unigrams are not expected to perform as well as bigrams (or 2-grams), as bigrams provide some concept of state for a given observation.

Features Selection

Section 3.4 discussed the concept of a *potential feature*. In the model search, we use a form of subset selection (section 2.2.1), creating an exhaustive list of feature combinations used to generate models, with 7 possible subsets:

1. [itime]
2. [packet type]
3. [payload histogram]
4. [itime, packet type]
5. [itime, payload histogram]
6. [packet type, payload histogram]
7. [itime, packet type, payload histogram]

Models generated from these lists of feature types can reveal what types of features provide useful information for predictions and those that are either redundant or wholly irrelevant.

Parameter Settings

Fundamentally, to perform a grid search, we must create a parameter space based on the parameters that define an OCSVM. For each parameter type, we select a finite set of "reasonable" values over which to search. We put emphasis on the term "reasonable" because, this likely many other settings, is somewhat arbitrary. ν must fall in the range (0, 1], γ is defined following recommendations of [32], and `degree` as applied to a polynomial kernel stays within comprehensible terms. For this experiment, we define the following parameter space:

```
ν = [0.01, 0.25, 0.5, 0.75, 1]
γ = [0.1, 0.01, 0.001, 0.0001]
degree = [1, 2, 3]
parameters = [{rbf,        ν, γ},
              {sigmoid,    ν, γ},
              {linear,     ν},
              {polynomial, ν, γ, degree}]
```

To generate the full parameter space, `sklearn.grid_search.GridSearchCV()` yields all combinations of parameters for any applicable ν, γ, and `degree` settings for each kernel option. Defining this parameter space results in 105 parameter settings to apply to OCSVM instances.

k-Fold CV

To understand how the trained OCSVM instances perform on unseen observations, we leverage k-fold CV with $k = 4$ folds across the n-gram observation set. With all other attribute settings constant, this 4-fold CV will generate four OCSVM model instances, each trained on 75% of the supplied n-gram observations and tested against the other 25% of observations. As k-fold CV is self-contained within `sklearn.grid_search.GridSearchCV()`, we ultimately issue commands to generate $\frac{5{,}880}{k} = 1{,}470$ OCSVMs.

TABLE 4.1: Attribute types and how their possible options contribute to the overall search space.

Attribute Type	Number of Options
n-grams	2
Features	7
Parameters	105
k-fold CV	4
Scoring Mechanisms	1
Model Instances	**5,880**

Scoring Mechanisms

In evaluating the performance of each model instance, the most viable scoring mechanism for an OCSVM is *accuracy*. An accuracy measurement provides a concrete value for the fraction of unseen observations that are correctly classified during each k-fold CV iteration. Additionally, we can directly calculate the FP rate of a model, $FP = 1 - accuracy$. Because OCSVMs only accept observations from a single class for training, any difference between 100% and the actual accuracy score is the FP rate for a given model, coming from untrained observations that are not correctly classified to the trained class.

4.2.2 Search Experiment Framework

Leveraging `sklearn.grid_search.GridSearchCV()`, each model in the grid search uses one CPU. Given both a large search space and large corpus, experimentation requires access to a great deal of CPU power. Running the search experiment on a Linux-based system S with access to 256 CPUs, it was possible to train 256 models concurrently. S's interaction interface forced the use of a slightly modified version of the USBeSafe model search algorithm, discussed further in Appendix D containing the USBeSafe project file hierarchy and descriptions.

Using the Python `scikit_learn.grid_search` library, we individually invoke `GridSearchCV()` as follows for each of the 1,470 attribute combinations:

```
clf = GridSearchCV(sklearn.svm.OneClassSVM(),
                   parameters, cv=4, n_jobs=1,
                   scoring="accuracy")
clf.fit(data, labels)
```

The grid search is first instantiated with the desired model type, a specific parameter setting from `parameters` defined in section 4.2.1, the number of folds for k-fold CV, the number of models to run concurrently (`n_jobs`), and the scoring mechanism. Because of S, the number of jobs run concurrently within the grid search framework must remain 1, though S can manage up to 256 of these individual grid searches. Next, the feature vectors are fit to the grid search instance; though OCSVM is an inherently unsupervised learning problem, the `sklearn.svm.OneClassSVM()` implementation requires labels for all observations. This issue is easily solved by

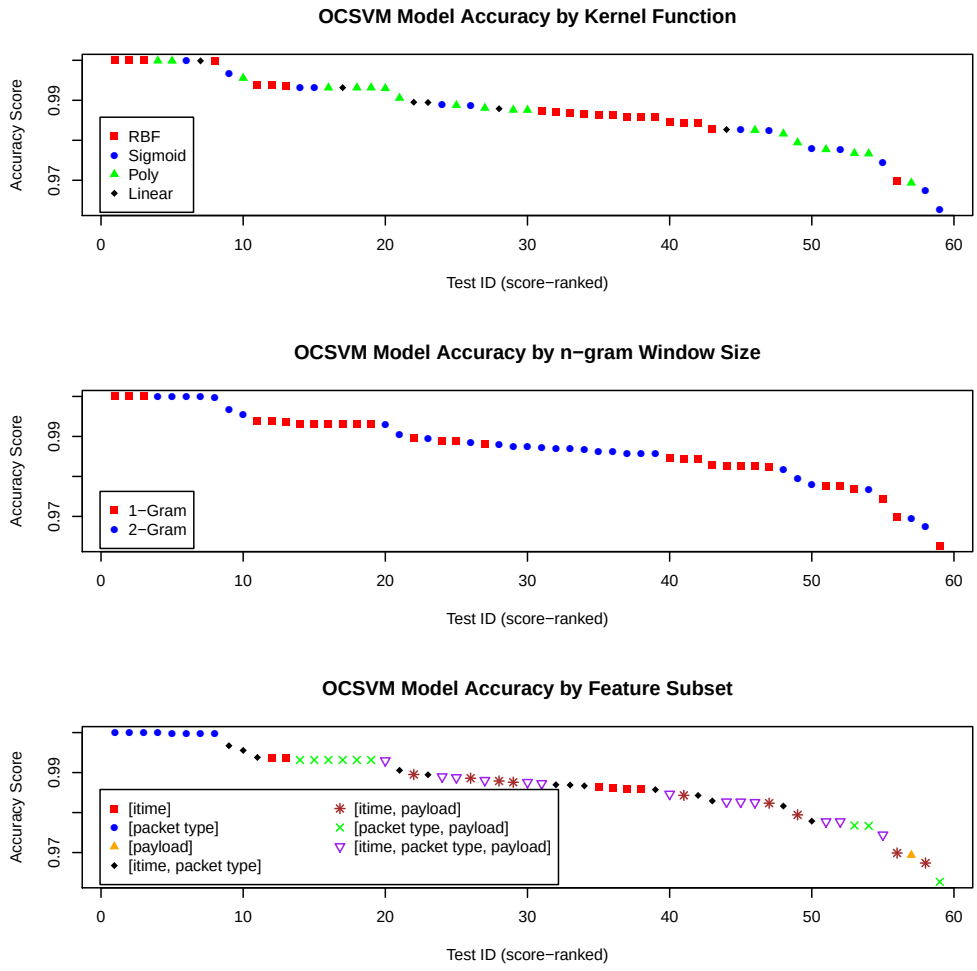

FIGURE 4.1: Model search results with an accuracy score threshold of 0.96 and independent test attributes.

supplying a list of 1's with an entry for each observation. This label set means that all observations are labeled +1, falling within the training set.

4.2.3 Experiment Results and Discussion

In total, 5,880 OCSVM instances were evaluated, when factoring in k-fold CV. As this functionality is internal to `GridSearchCV()`, we ultimately possess the ability to evaluate results from 1,470 OCSVM models, covering 105 parameter settings, two n-gram settings, and seven feature subsets, all scored according to an accuracy metric. The remainder of this section is a review and analysis of the results from this OCSVM grid search.

After the grid search was completed, further consideration was given to OCSVM models with resulting mean accuracy scores ≥ 0.96, corresponding to a FP rate of 4% or less. This threshold was determined according to previous SVM- and OVSVM-based NAD work [41, 58, 59, 62] in which most acceptable systems limited the FP rate to a slightly higher maximum of 4.5%. Of all 1,470 test models, 59 OCSVMs of varying attribute settings achieved this threshold.

TABLE 4.2: Evaluation of independent attributes' effects on OCSVM accuracy scores, with top 8 models removed.

Attribute	Attribute Option	Tests Scoring $\geq$ 0.96	Average Accuracy
Kernel	RBF	17	0.986171
	sigmoid	12	0.983500
	poly	17	0.985564
	linear	5	0.988538
Window	1	27	0.985220
	2	24	0.985287
Feature Subset	[itime]	6	0.988603
	[packet type]	0	N/A
	[payload]	1	0.969350
	[itime, packet type]	13	0.987611
	[itime, payload]	9	0.981880
	[packet type, payload]	9	0.986112
	[itime, packet type, payload]	13	0.984307

Figure 4.1 presents the results of the experiment after this threshold cut, independently based on varying attribute types: kernel, n-gram window size, and feature subset, respectively. We see a grouping of eight models, all with nearly 100% accuracy in each graph. Figure 4.1 reveals that these OCSVM instances all stem from the [packet type] feature subset. Though high accuracy is a major goal for model performance, such consistently high scores, regardless of kernel type or window size, indicates this subset may not yield viable candidates for OCSVM training. The reason for this is because the packet type feature can only take on eight distinct values (section 3.5.1). When training solely on packet type, each observation will equal one of these values. The domain is simply too small to accurately characterize USB traffic and leads to a system which can prove rather simple for an adversary to defeat. For this reason, we do not consider these model instances for further testing and are left with 51 possible models.

Table 4.2 summarizes the model search results based on independent attributes after these tests are removed from consideration. Using the 51 remaining OCSVM instances, a few more important observations can be made from Figure 4.1 and Table 4.2:

- The linear OCSVM kernel rather consistently underperforms versus the other kernels. Though it performs the same or better in some instances, there are a significantly greater number of tests based on the other three kernel functions that score ≥ 0.96. What this reveals about the input corpus is that, in general, its vectors are not easily linearly separable; this means that other kernel functions, forcing the data points into a higher dimensional space, achieve better hyperplane separation.

- Examining n-gram window size independent of other attribute types, we see that OCSVM models perform slightly better with an increased n. Though only 1- and 2-grams were tested, this result makes sense; as input data information loss decreases, prediction performance increases. Increased n provides more context for a given observation.

TABLE 4.3: Due to time constraints, a select few grid search tests could not be completed.

Test No.	OCSVM Parameters	n-Gram Window	Features
829	kernel = RBF ν = 0.75 γ = 0.01	2	payload
1336	kernel = polynomial ν = 0.75 γ = 0.1 degree = 2	2	packet type, payload
1354	kernel = polynomial ν = 0.75 γ = 0.0001 degree = 2	2	payload
1361	kernel = polynomial ν = 0.75 γ = 0.1 degree = 3	2	payload
1363	kernel = polynomial ν = 0.75 γ = 0.1 degree = 3	2	itime, payload
1364	kernel = polynomial ν = 0.75 γ = 0.1 degree = 3	2	packet type, payload
1365	kernel = polynomial ν = 0.75 γ = 0.1 degree = 3	2	itime, packet type, payload

- With regards to feature subsets, besides [itime] which scores above 0.99 for two different models and just below 0.99 in several other instances, all but one OCSVM need to learn at least two feature types to perform well. Test ID #57 (score-ranked) achieves 96.935% accuracy by learning payload histograms alone, but many models score better than #57, having learned more traffic characteristics.

After this examination, the decision was made to train these remaining 51 models that met the 96% accuracy threshold and did not overtly indicate they would be trivial for an attacker to overcome. In the final section of this chapter, we discuss the results from training on the corpus and individually testing these 51 models against a known malicious trace file.

Note that due to time constraints, we were not able to complete scoring results of seven OCSVM model instances. Descriptions of these models can be found in Table 4.3. Due to the large attribute space lending to significant test coverage and the grid search resulting in many high-scoring OCSVM models, we do not anticipate these unfinished tests would greatly add to our results.

4.3 Model Training and Testing

4.3.1 Model Training Experiment

Like the model search experiment, model training experiments were performed on S, capable of large-scale data processing with 256 CPUs and requiring a modified version of the USBeSafe training infrastructure. These modifications are addressed in Appendix D.

The 51 models addressed in section 4.2.3 were individually trained with applicable attributes on the corpus, containing 1,191,957 USB class code 0x03 (with included 0x00) packets observations.

4.3.2 Model Testing Experiment Framework

To test the performance of these 51 models against known malicious activity, we used an existing malicious trace file. This trace was created via previous work by Dr. Wil Robertson, *et al*. reverse engineering the BadUSB covert HID attack and contains USB keyboard traffic for opening a command prompt and executing a code injection attack. Each model, with requests found in *requests.train*, was loaded and tested against a file of the format *ngrams.preprocessed*, respective to the model's n-gram setting, containing the malicious traffic. For each model, `sklearn.svm.OneClassSVM.predict()` was invoked and classified the 541 input observations from the malicious trace. The tests were run in two iterations, one for each n-gram size, with results combined for further analysis.

4.3.3 Model Testing Results

Like the grid search experiment in section 4.2.3, it is useful to examine novel observation score results per OCSVM independently based on varying attribute types: kernel, n-gram window size, and feature subset. These distinctions are shown in Figure 4.2, with tests sorted according to descending novel observation score. This score represents the fraction of observations that were classified as novel or anomalous, assigning them to the -1 class, or outside the trained class. Table 4.4 lists the Pearson correlation coefficients for possible feature subsets against the novel observation score.

A number of important observations can be made from these results:

- One model, using the polynomial kernel trained on 2-grams from all three feature types classifies a significantly greater fraction of observations as anomalous versus every other test. This OCSVM classifies 71.7% of observations from the malicious trace file as novel, while the next closest model drops quickly to a 50.4% novel classification rate.
- Overall, kernel type used does not seem to indicate any drastic increase in novel observations. Figure 4.3 shows that, while outliers exist, kernel function as an independent attribute typically performs in approximately the same novel observation rate range regardless of setting.
- Novel observation scores by models using 2-grams are generally considerably higher than by those using 1-grams, reinforcing the concept

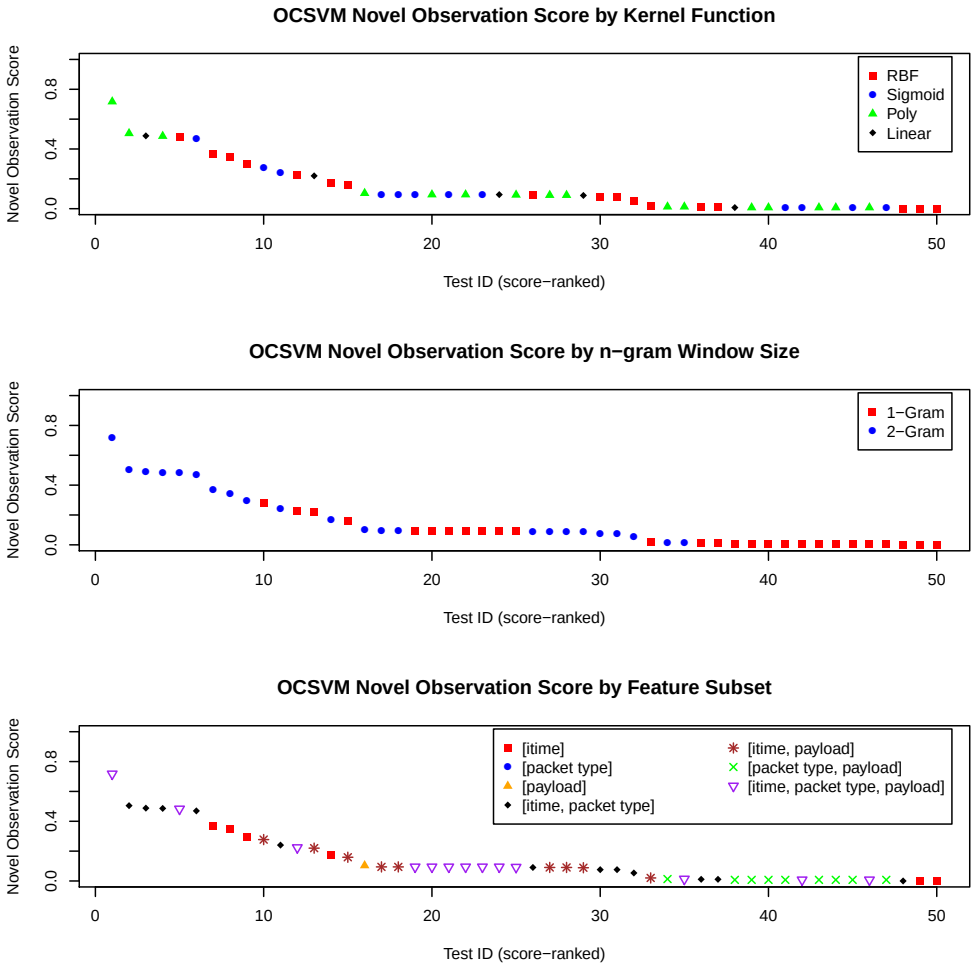

FIGURE 4.2: Model testing results against a known malicious trace using independent test attributes.

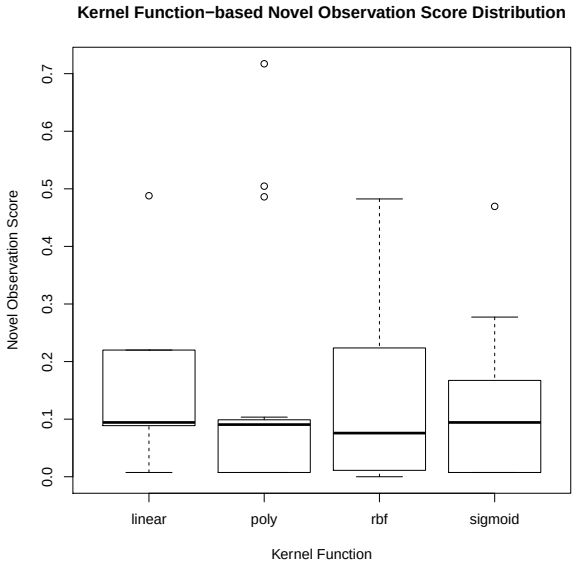

FIGURE 4.3: Novel observation scores from tests on a known malicious keyboard traffic trace, segregated by kernel function.

TABLE 4.4: Pearson correlation measures between feature subsets and novel observation scores.

Feature Subset	**Pearson Correlation to Novel Observation Score**
[itime]	0.3651734
[packet type]	-0.03712667
[payload]	-0.2761911
[itime, packet type]	0.2484312
[itime, payload]	0.02719254
[packet type, payload]	-0.2237014
[itime, packet type, payload]	0.06951541

that more context for any given observation provides better prediction performance. For all tests, the Pearson correlation coefficient between n and novel observation score is 0.5138353, indicating a fairly strong relationship between increased n and increased score. Nine of the top 10 classifiers, according to novel classification rate, are trained using 2-grams.

- Table 4.4 shows that there is a correlation between using itimes in a feature subset and an increase in novel observation score. For each feature subset that does not include itimes, when itimes are incorporated into the model, correlation increases.

It is also important to mention the possible threat of model overfitting. In Figure 4.4, we see a correlation between model accuracy scores during the grid search and the novel observation score during this testing phase; the Pearson correlation coefficient between these two variables is 0.1807714.

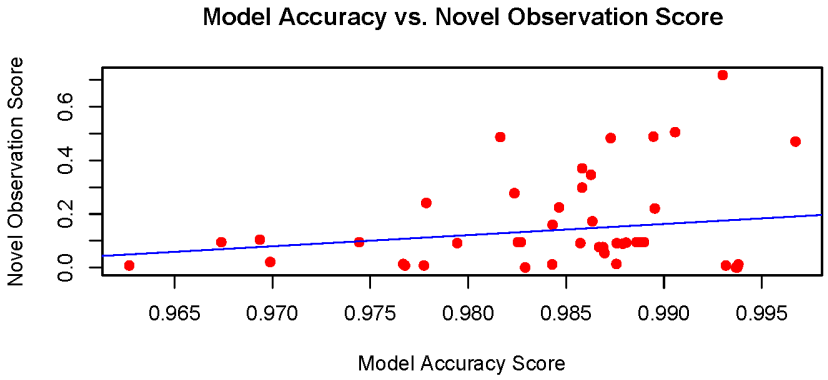

FIGURE 4.4: Though not necessarily strong, we see an obvious correlation between model accuracy and fraction of observations classified as novel.

In general, a higher accuracy score for classifying the single known class loosely translates to a higher percentage of new observations flagged as novel. For the system to minimize false positives and false negatives, it is valuable to find some balance between the model accuracy and the novel observation rate. We recognize that further testing on other known malicious trace files will serve to provide more information about which models continue to perform well as attack generalization increases.

Because we consider USBeSafe an offline USB event anomaly detection system serving as the basis for a live detection system, these results beg the question of when the system should flag activity as anomalous. There exist several options for defining alert thresholds:

- **Overall-score threshold:** For an offline system, it is possible to define an overall novel classification score threshold and alert if the score for a given input exceeds this threshold.
- **Consecutive-observation threshold:** For both offline and live systems, it is possible to set a threshold for the number of consecutive observations classified as anomalous before the system alerts on potentially malicious activity.
- **Window-fraction threshold:** For both offline and live systems, a threshold can be set for a given window size indicating the fraction of observations that must be classified as novel within that window before the system presents an alert.

A production version of USBeSafe would likely use a combination of these threshold metrics and leaves the door open for further research to improve the system.

This research reveals that there exist features specific to USB keyboard traffic that possess the ability to identify such traffic as benign or malicious, and it is possible to leverage those features to train an OCSVM capable of differentiating between a malicious code injection attack and benign keyboard traffic. We successfully applied OCSVM to model a large corpus of

benign USB keyboard traffic and tested viable models against a known malicious attack with instances of high novel observation rates. We also found significant correlations between inclusion of itime feature observations and novel observation scores as well as increased n-gram window size to novel observation score. We identified a subset of models from the search experiment that perform strongly against the malicious trace, and introduced an alert framework for both offline and live versions of USBeSafe.

Chapter 5

Conclusions and Future Work

In this chapter, we first present a final overview of USBeSafe and its contributions to the fields of USB security and anomaly detection. We then conclude by discussing a number of avenues for future research in the area of USB event anomaly detection.

5.1 Conclusions

After the introduction of BadUSB as a brand new attack class via USB, it seemed that protecting against malicious behavior built directly into device firmware would prove extremely difficult, if not impossible. The implications for a lack of protection in this area are enormous; with the number of USB devices in end-user computing continuing to rise, the surface area for BadUSB-style attacks grows each day. Protection mechanisms such as IEEE 1667 [69] and GoodUSB [76] have been proposed, but they have gained little to no mainstream traction and change the end-user operational status quo for USB devices.

Through USBeSafe, we show that it is possible to leverage ML techniques to successfully detect malicious USB traffic, thereby alleviating the need to involve the user in precautionary security measures. USBeSafe, in its current form, serves as a means of detecting a rogue-TD attack in which a covert keyboard interface is defined and operating in the device firmware.

The development of USBeSafe contributes significant progress to USB security when considering the threat from rogue-TD attacks. Through this work, we accomplished the following:

1. Leveraged the Linux kernel module `usbmon` to create and characterize a corpus of USB device traffic from the perspective of the bus.

2. Achieved efficient extraction of USB traffic features and provided an extensible framework for expanding the types of features and USB device classes considered.

3. First-of-its-kind application of OCSVM to USB keyboard traffic features, yielding accurate identification of benign keyboard traffic.

4. Completed comprehensive feature and model attribute selection for USB keyboard traffic with analysis of feature and attribute relevance specific to the corpus.

5. Provided a platform for applying OCSVM to both offline and live USBeSafe USB traffic anomaly detection systems.

There do exist some limitations to this research. First, we are required to assume that the traffic corpus is wholly benign, containing no malicious traffic whatsoever. This assumption is what allows us to leverage OCSVM; though unlikely, the possibility of malicious traffic in the USBeSafe corpus would lend to the use of a standard supervised learning-based SVM with binary classification capabilities. We also recognize that a single malicious attack test is not necessarily enough to indicate the viability of the approach or a specific OCSVM model. Development of any new attack trace is simply a slight variation of the existing command injection attack trace and may not necessarily prove useful. Lastly, we consider a fundamental shortcoming of anomaly detection: the potential for a *mimicry attack* in which an adversary forms an attack input in such a way that it looks normal to the anomaly detection mechanism [60]. We recognize this shortfall, especially with regards to USB packet itimes. Because many models that result in high novel observation scores incorporate itimes, the potential exists for an adversary to craft an attack that does match human typing patterns. Fortunately, the overhead in developing malicious firmware for such an attack is significantly greater than existing attack examples.

5.2 Future Work

While USBeSafe shows significant promise in detecting anomalous USB traffic, there are many areas for future research in this field. Due to its flexible nature, the rogue-TD attack vector possesses significant potency; there exist a number of options to explore as to furthering the capabilities of USBeSafe as well exploring the technical shortcomings of other solutions.

Specific to our USBeSafe model search experiments, it is possible to examine how larger window size for n-grams affect OCSVM performance; as we see an increase in accuracy from $n = 1$ to $n = 2$ (albeit small, but measured independently from other attribute factors), increasing n further may prove fruitful in terms of accuracy scores and detection performance by providing more state and context for a given observation.

Another useful avenue of work would be to explore the generalizability of the keyboard emulation attack detection mechanism. By gathering a larger corpus from many users, we could determine if keyboard traffic models are universally applicable or should be tuned on an individual basis.

We can also explore options for detecting the other types of attacks demonstrated in BadUSB. For example, how can we leverage SVM technology to effectively characterize the network spoof attack? Such a problem may not involve only USB traffic; we may have to leverage and model information found in system change logs pertinent to network configurations.

Appendix A

USB Descriptors

Each table below [12] serves as a USB descriptor byte map, outlining the contents of each descriptor type. The tables are listed in hierarchical order with the device descriptor at the top of the descriptor hierarchy, as this is the first descriptor the host receives from a device.

TABLE A.1: Contents of a device descriptor packet, by byte [12].

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of the Descriptor in Bytes (18 bytes)
1	bDescriptorType	1	Constant	Device Descriptor (0x01)
2	bcdUSB	2	BCD	USB Specification Number which device complies too.
4	bDeviceClass	1	Class	Class Code (Assigned by USB Org) 0x00: each interface specifies class code. 0xFF: the class code is vendor specified. Otherwise field is valid Class Code.
5	bDeviceSubClass	1	SubClass	Subclass Code (Assigned by USB Org)
6	bDeviceProtocol	1	Protocol	Protocol Code (Assigned by USB Org)
7	bMaxPacketSize	1	Number	Maximum Packet Size for Zero Endpoint. Valid Sizes are 8, 16, 32, 64
8	idVendor	2	ID	Vendor ID (Assigned by USB Org)
10	idProduct	2	ID	Product ID (Assigned by Manufacturer)
12	bcdDevice	2	BCD	Device Release Number
14	iManufacturer	1	Index	Index of Manufacturer String Descriptor
15	iProduct	1	Index	Index of Product String Descriptor
16	iSerialNumber	1	Index	Index of Serial Number String Descriptor
17	bNumConfigurations	1	Integer	Number of Possible Configurations

TABLE A.2: Contents of a configuration descriptor packet, by byte [12].

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes
1	bDescriptorType	1	Constant	Configuration Descriptor (0x02)
2	wTotalLength	2	Number	Total length in bytes of data returned
4	bNumInterfaces	1	Number	Number of Interfaces
5	bConfigurationValue	1	Number	Value to use as an argument to select this configuration
6	iConfiguration	1	Index	Index of String Descriptor describing this configuration
7	bmAttributes	1	Bitmap	D7 Reserved, set to 1. (USB 1.0 Bus Powered) D6 Self Powered D5 Remote Wakeup D4..0 Reserved, set to 0.
8	bMaxPower	1	mA	Maximum Power Consumption in 2mA units

TABLE A.3: Contents of an interface descriptor packet, by byte [12].

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes (9 Bytes)
1	bDescriptorType	1	Constant	Interface Descriptor (0x04)
2	bInterfaceNumber	1	Number	Number of Interface
3	bAlternateSetting	1	Number	Value used to select alternative setting
4	bNumEndpoints	1	Number	Number of Endpoints used for this interface
5	bInterfaceClass	1	Class	Class Code (Assigned by USB Org)
6	bInterfaceSubClass	1	SubClass	Subclass Code (Assigned by USB Org)
7	bInterfaceProtocol	1	Protocol	Protocol Code (Assigned by USB Org)
8	iInterface	1	Index	Index of String Descriptor Describing this interface

TABLE A.4: Contents of an endpoint descriptor packet, by byte [12].

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes (7 bytes)
1	bDescriptorType	1	Constant	Endpoint Descriptor (0x05)
2	bEndpointAddress	1	Endpoint	Endpoint Address Bits 0..3b Endpoint Number. Bits 4..6b Reserved. Set to Zero Bits 7 Direction 0 = Out, 1 = In (Ignored for Control Endpoints)
3	bmAttributes	1	Bitmap	Bits 0..1 Transfer Type 00 = Control 01 = Isochronous 10 = Bulk 11 = Interrupt Bits 2..7 are reserved. If Isochronous endpoint, Bits 3..2 = Synchronisation Type (Iso Mode) 00 = No Synchronisation 01 = Asynchronous 10 = Adaptive 11 = Synchronous Bits 5..4 = Usage Type (Iso Mode) 00 = Data Endpoint 01 = Feedback Endpoint 10 = Explicit Feedback Data Endpoint 11 = Reserved
4	wMaxPacketSize	2	Number	Maximum Packet Size this endpoint is capable of sending or receiving
6	bInterval	1	Number	Interval for polling endpoint data transfers. Value in frame counts. Ignored for Bulk & Control Endpoints. Isochronous must equal 1; field may range [1, 255] for interrupt endpoints.
6	bInterfaceSubClass	1	SubClass	Subclass Code (Assigned by USB Org)
7	bInterfaceProtocol	1	Protocol	Protocol Code (Assigned by USB Org)
8	iInterface	1	Index	Index of String Descriptor Describing this interface

TABLE A.5: Contents of a string descriptor packet, by byte [12].

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes
1	bDescriptorType	1	Constant	String Descriptor (0x03)
2	wLANGID[0]	2	number	Supported Language Code Zero (e.g. 0x0409 English - United States)
4	wLANGID[1]	2	number	Supported Language Code One (e.g. 0x0c09 English - Australian)
n	wLANGID[x]	2	number	Supported Language Code x (e.g. 0x0407 German - Standard)

Appendix B

Interarrival Time Histograms by `pause` Length

When determining session length to mitigate issues with potential infinitely long itimes between packets (section 3.4.2), we considered three possible `pause` values to determine the session cutoff: 20000ms, 40000ms, and 60000ms. Manual examination of these normalized itimes involved graphing them in a number of histograms with varying bin intervals. After an initial search, we find that `pause` length has little impact on model performance and choose to process itimes according to `pause` = 20000ms.

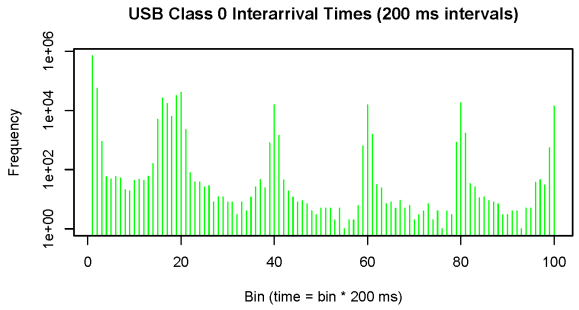

FIGURE B.1: Histogram showing interarrival times of USB class code 0x00, normalized on a 20000ms `pause` with bin intervals of 200ms.

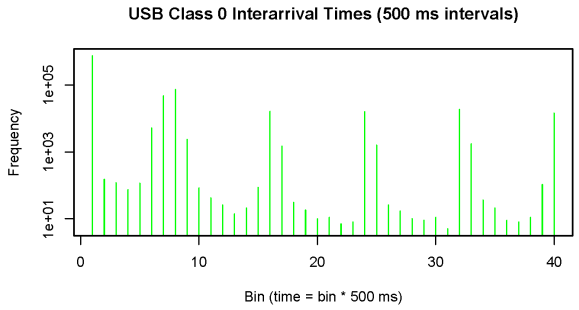

FIGURE B.2: Histogram showing interarrival times of USB class code 0x00, normalized on a 20000ms `pause` with bin intervals of 500ms.

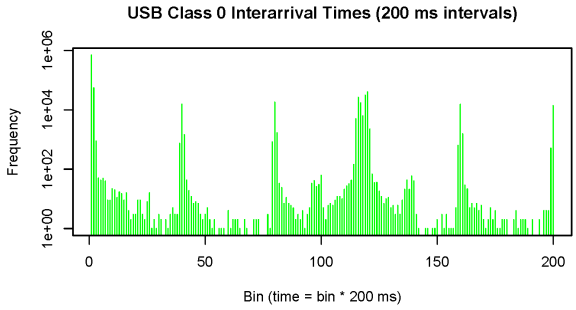

FIGURE B.3: Histogram showing interarrival times of USB class code 0x00, normalized on a 40000ms `pause` with bin intervals of 200ms.

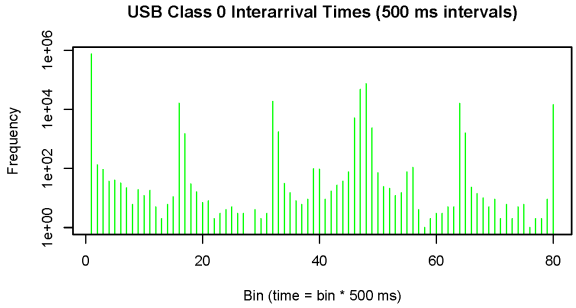

FIGURE B.4: Histogram showing interarrival times of USB class code 0x00, normalized on a 40000ms `pause` with bin intervals of 500ms.

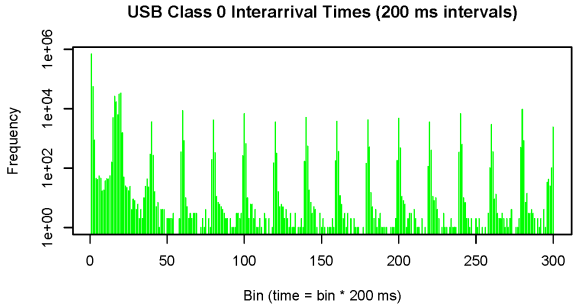

FIGURE B.5: Histogram showing interarrival times of USB class code 0x00, normalized on a 60000ms `pause` with bin intervals of 200ms.

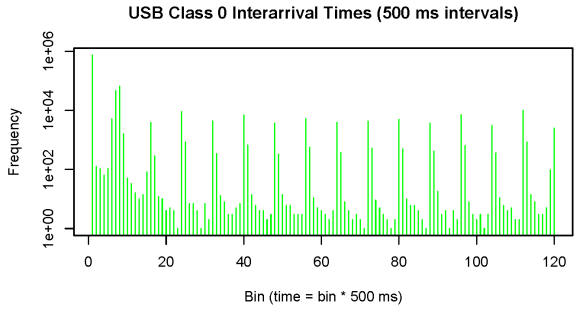

FIGURE B.6: Histogram showing interarrival times of USB class code 0x00, normalized on a 60000ms `pause` with bin intervals of 500ms.

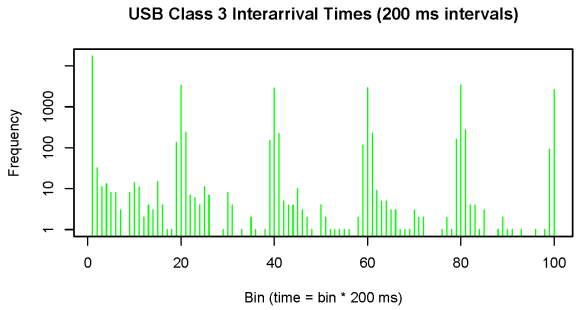

FIGURE B.7: Histogram showing interarrival times of USB class code 0x03, normalized on a 20000ms `pause` with bin intervals of 200ms.

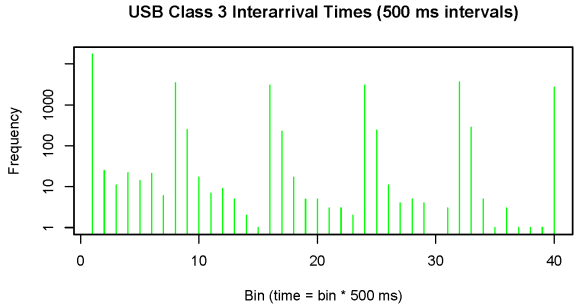

FIGURE B.8: Histogram showing interarrival times of USB class code 0x03, normalized on a 20000ms `pause` with bin intervals of 500ms.

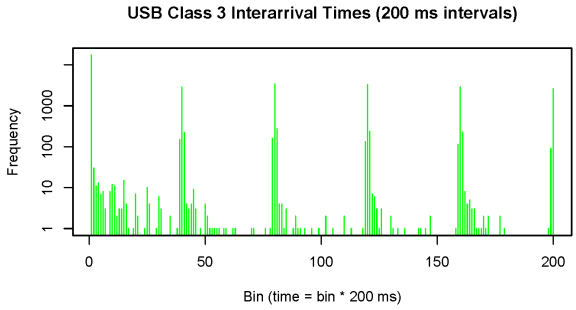

FIGURE B.9: Histogram showing interarrival times of USB class code 0x03, normalized on a 40000ms `pause` with bin intervals of 200ms.

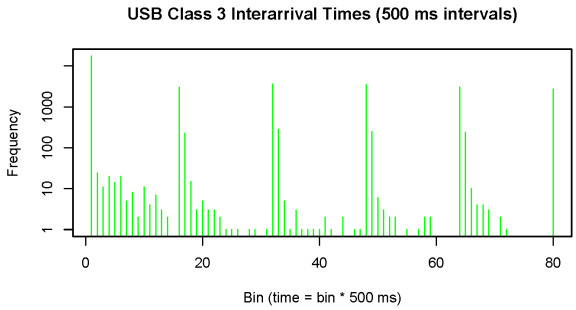

FIGURE B.10: Histogram showing interarrival times of USB class code 0x03, normalized on a 40000ms `pause` with bin intervals of 500ms.

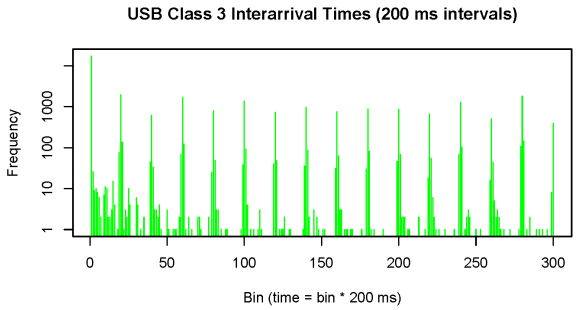

FIGURE B.11: Histogram showing interarrival times of USB class code 0x03, normalized on a 60000ms `pause` with bin intervals of 200ms.

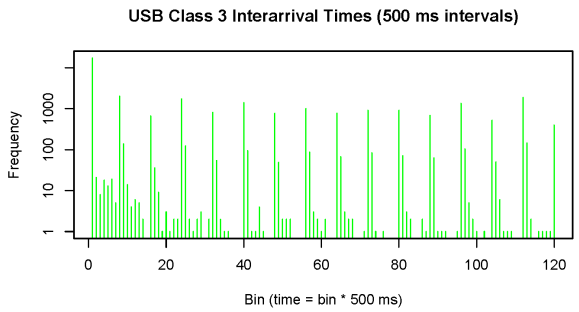

FIGURE B.12: Histogram showing interarrival times of USB class code 0x03, normalized on a 60000ms `pause` with bin intervals of 500ms.

Appendix C

Model Training Requests

During model training in section 4.3, 51 models were generated according to the below parameters. The file *requests.train* contains the below model requests for training, each formatted as follows:

(USB class code, [feature types], {OCSVM parameters}, n-gram window size)

The 51 model requests are as follows:

```
(3, ['itime', 'ptype'],
    {'kernel': 'sigmoid', 'nu': 0.01, 'gamma': 0.01}, 2)
(3, ['itime', 'ptype'],
    {'kernel': 'poly', 'nu': 0.01, 'gamma': 0.001, 'degree': 1}, 2)
(3, ['itime', 'ptype'],
    {'kernel': 'rbf', 'nu': 0.01, 'gamma': 0.1}, 1)
(3, ['itime'], {'kernel': 'rbf', 'nu': 0.01, 'gamma': 0.0001}, 1)
(3, ['itime'], {'kernel': 'rbf', 'nu': 0.01, 'gamma': 0.01}, 1)
(3, ['ptype', 'payload'],
    {'kernel': 'sigmoid', 'nu': 0.01, 'gamma': 0.0001}, 1)
(3, ['ptype', 'payload'],
    {'kernel': 'sigmoid', 'nu': 0.01, 'gamma': 0.001}, 1)
(3, ['ptype', 'payload'],
    {'kernel': 'poly', 'nu': 0.01, 'gamma': 0.1, 'degree': 1}, 1)
(3, ['ptype', 'payload'],
    {'kernel': 'linear', 'nu': 0.01}, 1)
(3, ['ptype', 'payload'],
    {'kernel': 'poly', 'nu': 0.01, 'gamma': 0.0001, 'degree': 1}, 1)
(3, ['ptype', 'payload'],
    {'kernel': 'poly', 'nu': 0.01, 'gamma': 0.001, 'degree': 1}, 1)
(3, ['itime', 'ptype', 'payload'],
    {'kernel': 'poly', 'nu': 0.01, 'gamma': 0.0001, 'degree': 3}, 2)
(3, ['itime', 'ptype'],
    {'kernel': 'poly', 'nu': 0.01, 'gamma': 0.001, 'degree': 1}, 2)
(3, ['itime', 'payload'],
    {'kernel': 'linear', 'nu': 0.01}, 1)
(3, ['itime', 'ptype'],
    {'kernel': 'linear', 'nu': 0.01}, 2)
(3, ['itime', 'ptype', 'payload'],
    {'kernel': 'sigmoid', 'nu': 0.01, 'gamma': 0.001}, 1)
(3, ['itime', 'ptype', 'payload'],
    {'kernel': 'poly', 'nu': 0.01, 'gamma': 0.01, 'degree': 1}, 1)
(3, ['itime', 'payload'],
```

```
    {'kernel': 'sigmoid', 'nu': 0.01, 'gamma': 0.1}, 2)
(3, ['itime', 'ptype', 'payload'],
    {'kernel': 'poly', 'nu': 0.01, 'gamma': 0.001, 'degree': 3}, 1)
(3, ['itime', 'payload'],
    {'kernel': 'linear', 'nu': 0.01}, 2)
(3, ['itime', 'payload'],
    {'kernel': 'poly', 'nu': 0.01, 'gamma': 0.1, 'degree': 1}, 2)
(3, ['itime', 'ptype', 'payload'],
    {'kernel': 'poly', 'nu': 0.01, 'gamma': 0.001, 'degree': 3}, 2)
(3, ['itime', 'ptype', 'payload'],
    {'kernel': 'rbf', 'nu': 0.01, 'gamma': 0.01}, 2)
(3, ['itime', 'ptype'],
    {'kernel': 'rbf', 'nu': 0.01, 'gamma': 0.0001}, 2)
(3, ['itime', 'ptype'],
    {'kernel': 'rbf', 'nu': 0.01, 'gamma': 0.001}, 2)
(3, ['itime', 'ptype'],
    {'kernel': 'rbf', 'nu': 0.01, 'gamma': 0.01}, 2)
(3, ['itime'],
    {'kernel': 'rbf', 'nu': 0.01, 'gamma': 0.0001}, 2)
(3, ['itime'],
    {'kernel': 'rbf', 'nu': 0.01, 'gamma': 0.01}, 2)
(3, ['itime'],
    {'kernel': 'rbf', 'nu': 0.01, 'gamma': 0.1}, 2)
(3, ['itime'],
    {'kernel': 'rbf', 'nu': 0.01, 'gamma': 0.001}, 2)
(3, ['itime', 'ptype'],
    {'kernel': 'rbf', 'nu': 0.01, 'gamma': 0.1}, 2)
(3, ['itime', 'ptype', 'payload'],
    {'kernel': 'rbf', 'nu': 0.01, 'gamma': 0.01}, 1)
(3, ['itime', 'payload'],
    {'kernel': 'rbf', 'nu': 0.01, 'gamma': 0.01}, 1)
(3, ['itime', 'ptype'],
    {'kernel': 'rbf', 'nu': 0.01, 'gamma': 0.01}, 1)
(3, ['itime', 'ptype'],
    {'kernel': 'rbf', 'nu': 0.01, 'gamma': 0.0001}, 1)
(3, ['itime', 'ptype', 'payload'],
    {'kernel': 'linear', 'nu': 0.01}, 1)
(3, ['itime', 'ptype', 'payload'],
    {'kernel': 'sigmoid', 'nu': 0.01, 'gamma': 0.1}, 1)
(3, ['itime', 'ptype', 'payload'],
    {'kernel': 'poly', 'nu': 0.01, 'gamma': 0.1, 'degree': 1}, 1)
(3, ['itime', 'payload'],
    {'kernel': 'sigmoid', 'nu': 0.01, 'gamma': 0.1}, 1)
(3, ['itime', 'ptype'],
    {'kernel': 'poly', 'nu': 0.01, 'gamma': 0.1, 'degree': 2}, 2)
(3, ['itime', 'payload'],
    {'kernel': 'poly', 'nu': 0.01, 'gamma': 0.01, 'degree': 1}, 2)
(3, ['itime', 'ptype'],
    {'kernel': 'sigmoid', 'nu': 0.01, 'gamma': 0.1}, 2)
(3, ['itime', 'ptype', 'payload'],
    {'kernel': 'poly', 'nu': 0.01, 'gamma': 0.0001, 'degree': 1}, 1)
```

```
(3, ['itime', 'ptype', 'payload'],
    {'kernel': 'sigmoid', 'nu': 0.01, 'gamma': 0.0001}, 1)
(3, ['ptype', 'payload'],
    {'kernel': 'poly', 'nu': 0.01, 'gamma': 0.01, 'degree': 1}, 1)
(3, ['ptype', 'payload'],
    {'kernel': 'poly', 'nu': 0.01, 'gamma': 0.001, 'degree': 3}, 2)
(3, ['itime', 'ptype', 'payload'],
    {'kernel': 'sigmoid', 'nu': 0.01, 'gamma': 0.01}, 1)
(3, ['itime', 'payload'],
    {'kernel': 'rbf', 'nu': 0.01, 'gamma': 0.001}, 1)
(3, ['payload'],
    {'kernel': 'poly', 'nu': 0.01, 'gamma': 0.0001, 'degree': 3}, 2)
(3, ['itime', 'payload'],
    {'kernel': 'sigmoid', 'nu': 0.01, 'gamma': 0.01}, 2)
(3, ['ptype', 'payload'],
    {'kernel': 'sigmoid', 'nu': 0.01, 'gamma': 0.01}, 1)
```

Appendix D

USBeSafe Project Files

Appendix D contains the USBeSafe project hierarchy and directory/file descriptions. * serves as a wildcard character to condense similar files.

D.1 Project Hierarchy

```
USBeSafe
├── attack
│   ├── 1gram_vectors_20160420-110018.preprocessed
│   ├── 2gram_vectors_20160420-110026.preprocessed
│   ├── features_20160420-105955.extracted
│   └── keyboard_filtered.pcap
└── experiments
    ├── search
    │   ├── generate_model.py
    │   ├── get_scores.bash
    │   ├── llgrid_search.bash
    │   ├── ngram_file_list.txt
    │   ├── parameter_space.txt
    │   ├── process_scores.py
    │   ├── search_requests
    │   │   └── search_reqs_*.json
    │   ├── search_results.csv
    │   ├── test_logs
    │   │   └── test_*.bash.log-*
    │   └── test_scripts
    │       └── test_*.bash
    └── train
        ├── llgrid_train.bash
        ├── requests.train
        ├── training_info
        │   └── class_3
        │       └── model_*.OCSVM*
        ├── train_logs
        │   └── train_*.bash.log-*
        ├── train_outputs
        │   ├── model_*.trained
        │   └── models.trained
        ├── train.py
        └── train_scripts
            └── train_*.bash
```

```
├── models.trained
│   ├── models_1gram.trained
│   └── models_2gram.trained
├── ngrams.preprocessed
│   ├── 1gram_vectors_20160330-150842.preprocessed
│   ├── 2gram_vectors_20160330-151003.preprocessed
│   ├── 4gram_vectors_20160330-151235.preprocessed
│   └── 8gram_vectors_20160330-151633.preprocessed
├── results.tested
│   ├── results_20160420-114649.csv
│   ├── results_20160420-114725.csv
│   └── results_combined.csv
├── supporting
│   ├── R_data
│   │   ├── itime_histograms
│   │   │   └── class_*_itime_*_ms_plots.pdf
│   │   ├── itime_histograms.r
│   │   ├── plot_search_results.r
│   │   ├── plot_test_results.r
│   │   └── search_vs_test.r
│   └── search_vs_test.csv
└── usb.py
```

D.2 Directory and File Descriptions

For directory and file descriptions, directory names are labeled in red and file names are labeled in blue.

attack: contains all files related to the malicious trace *keyboard.pcap* applied against various trained models in section 4.3.2

1gram_vectors_20160420-110018.preprocessed: 1-gram attack feature vectors used for testing against a given model

2gram_vectors_20160420-110026.preprocessed: 2-gram attack feature vectors used for testing against a given model

features_20160420-105955.extracted: extracted attack features database

keyboard_filtered.pcap: a filtered `PCAP` file of the keyboard attack containing packets from Bus 0x03, Address 0x03 used as input for the malicious input experiment in 4.3.2

experiments: contains the framework for the grid search and training experiments

search: contains grid search experiment framework and auxiliary information

generate_model.py: Python script leveraged by *llgrid_search.bash* to generate a single model according to supplied attributes

get_scores.bash: pulls score information for training experiment post-processing from *test_*.bash.log-**; leveraged by *process_scores.py*

llgrid_search.bash: grid search experiment script that creates and invokes all *test_*.bash* with associated log information in respective *test_*.bash.log-**

ngram_file_list.txt: contains pathnames to the n-gram feature vector files used in the grid search; leveraged by *llgrid_search.bash*

parameter_space.txt: contains all variations of OCSVM parameters for use in the grid search; leveraged by *llgrid_search.bash*

process_scores.py: Python script used to generate overall score information from grid search experiment and creates *search_results.csv* as output

search_requests: contains feature subset requests as attribute for grid search

search_reqs_*.json: files containing individual requests for all feature subsets for a grid search; leveraged by *llgrid_search.bash*

search_results.csv: generated by *process_scores.py* and contains scoring information for each model with associated attributes sorted from high to low accuracy score

test_logs: contains the logs generated from all 1,470 grid search tests

test_*.bash.log-*: each log, 1,470 in all, contains output from each OCSVM instance in the grid search; generated by *llgrid_search.bash*

test_scripts: contains scripts generated to invoke all 1,470 grid search tests

test_*.bash: each script, 1,470 in all, invokes an OCSVM instance in the grid search attribute space; generated by *llgrid_search.bash*

train: contains the model training experiment framework and auxiliary information

llgrid_train.bash: model training experiment script that creates and invokes *train_*.bash* with associated log information in respective *train_*.bash.log-**

requests.train: listing of models to train according to associated attributes during the model training experiment

training_info: contains trained model files sorted by USB class code

class_3: contains trained model files for USB class code 0x03 traffic

model_*.OCSVM*: various models according to grid search test ID, with

model_*.OCSVM serving as the base file for loading a given model and *model_*.OCSVM** acting as supporting model files, 10 for each model

train_logs: contains the logs generated from 51 OCSVM training instances

train_*.bash.log-*: each log, 51 in all, contains output from each OCSVM instance in the model training experiment; generated by *llgrid_train.bash*

train_outputs: contains model training outputs from *train.py*

model_*.trained: each file, 51 in all, contains attribute and filepath information for the trained model with which it is associated

models.trained: file containing all data from *model_*.trained*

train.py: Python script leveraged by *llgrid_train.bash* to generate a single model according supplied attributes

train_scripts: contains scripts generated to invoke each of 51 model training instances

train_*.bash: each script, 51 in all, invokes the training of an OCSVM instance in the model training experiment; generated by *llgrid_train.bash*

models.trained: contains n-gram based listings of models trained during the training experiment by *llgrid_train.bash*

models_1gram.trained: listing of model file paths trained on 1-grams by *llgrid_train.bash*; used as input for testing in 4.3.2

models_2gram.trained: listing of model file paths trained on 2-grams by *llgrid_train.bash*; used as input for testing in 4.3.2

ngrams.preprocessed: contains n-gram vector files generated from USB traffic corpus

1gram_vectors_20160330-150842.preprocessed: 1-gram feature vector database for the USB traffic corpus, sorted by USB class code

2gram_vectors_20160330-151003.preprocessed: 2-gram feature vector database for the USB traffic corpus, sorted by USB class code

4gram_vectors_20160330-151235.preprocessed: 4-gram feature vector database for the USB traffic corpus, sorted by USB class code

8gram_vectors_20160330-151633.preprocessed: 8-gram feature vector database for the USB traffic corpus, sorted by USB class code

results.tested: contains results from model testing against *keyboard_filtered.pcap*

results_20160420-114649.csv: 1-gram model testing results against *keyboard_filtered.pcap*

results_20160420-114725.csv: 2-gram model testing results against *keyboard_filtered.pcap*

results_combined.csv: combined *results_20160420-114649.csv* and *results_20160420-114725.csv*, sorted on score from high to low

supporting: contains supporting scripts and outputs for analysis of results during experiments

R_data: contains supporting R scripts and various outputs used in statistical result analysis

itime_histograms: contains the histograms generated to examine the effect of various `pause` lengths on itime distribution

class_*_itime_*_ms_plots.pdf: series of histograms with varying USB class codes that show the effective distribution of itimes when `pause` and bin size vary

itime_histograms.r: R script that plots *class_*_itime_*_ms_plots.pdf*

plot_search_results.r: R script that plots model accuracy scores for each model from the grid search with independent model attribute scope

plot_test_results.r: R script that plots novel observation scores for each model tested against the malicious trace with independent model attribute scope

search_vs_test.r: R script that plots the model accuracy scores vs. novel observation scores

search_vs_test.csv: a manually constructed comparison of grid search accuracy scores vs. novel observation scores for the trained models

usb.py: self-contained USBeSafe Python codebase

Bibliography

[1] GIA. *3 Billion USB 3.0 Devices by 2018 – Global Industry Analysts*. 2013.

[2] Noah Shachtman. "Under Worm Assault, Military Bans Disks, USB Drives". In: *Wired, Nov* (2008).

[3] David E Sanger. *Confront and conceal: Obama's secret wars and surprising use of American power*. Crown Pub, 2012.

[4] David E Sanger. "Obama order sped up wave of cyberattacks against Iran". In: *The New York Times* 1.06 (2012), p. 2012.

[5] Phil Muncaster. *Indian navy computers stormed by malware-ridden USBs*. 2012.

[6] Ponemon. *2011 Second Annual Cost of Cyber Crime Study Benchmark Study of U.S. Companies*. 2011.

[7] Karsten Nohl and Jakob Lell. "BadUSB–On accessories that turn evil". In: *Black Hat USA* (2014).

[8] K. Nohl, S. Krißler, and J. Lell. *Turning USB peripherals into BadUSB*. 2014.

[9] Jan Axelson. *USB complete: the developer's guide*. Lakeview research LLC, 2015.

[10] Abhishek Gupta. *USB 3.0 vs USB 2.0: A quick reference summary for the busy engineer*. Cypress Semiconductor, 2014.

[11] HP. *USB 3.0 Technology: Performance Advantage on HP Workstations*. 2012.

[12] Craig Peacock. "USB in a nutshell. Making sense of the USB standard". In: (2002).

[13] Microsoft. *USB device class drivers included in Windows*. 2016.

[14] Phil Simon. *Too Big to Ignore: The Business Case for Big Data*. Vol. 72. John Wiley & Sons, 2013.

[15] Thomas M Mitchell et al. *Machine learning*. 1997.

[16] Christopher M Bishop. "Pattern Recognition". In: *Machine Learning* (2006).

[17] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2012.

[18] Bo Pang, Lillian Lee, and Shivakumar Vaithyanathan. "Thumbs up?: sentiment classification using machine learning techniques". In: *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*. Association for Computational Linguistics. 2002, pp. 79–86.

[19] Thiago S Guzella and Walmir M Caminhas. "A review of machine learning approaches to spam filtering". In: *Expert Systems with Applications* 36.7 (2009), pp. 10206–10222.

[20] Simon Tong and Edward Chang. "Support vector machine active learning for image retrieval". In: *Proceedings of the ninth ACM international conference on Multimedia*. ACM. 2001, pp. 107–118.

[21] Adam Coates et al. "Text detection and character recognition in scene images with unsupervised feature learning". In: *Document Analysis and Recognition (ICDAR), 2011 International Conference on*. IEEE. 2011, pp. 440–445.

[22] Geoffrey Hinton et al. "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups". In: *Signal Processing Magazine, IEEE* 29.6 (2012), pp. 82–97.

[23] Sachin Agarwal and Shilpa Arora. "Context based word prediction for texting language". In: *Large Scale Semantic Access to Content (Text, Image, Video, and Sound)*. LE CENTRE DE HAUTES ETUDES INTERNATIONALES D'INFORMATIQUE DOCUMENTAIRE. 2007, pp. 360–368.

[24] Yingbo Song, Angelos D Keromytis, and Salvatore Stolfo. "Spectrogram: A mixture-of-markov-chains model for anomaly detection in web traffic". In: *Network and Distributed System Security Symposium 2009: February 8-11, 2009, San Diego, California: Proceedings*. Internet Society. 2009, pp. 121–135.

[25] Gustavo EAPA Batista and Maria Carolina Monard. "An analysis of four missing data treatment methods for supervised learning". In: *Applied Artificial Intelligence* 17.5-6 (2003), pp. 519–533.

[26] Pat Langley et al. *Selection of relevant features in machine learning*. Defense Technical Information Center, 1994.

[27] Sotiris B Kotsiantis, I Zaharakis, and P Pintelas. *Supervised machine learning: A review of classification techniques*. 2007.

[28] Hanchuan Peng, Fuhui Long, and Chris Ding. "Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 27.8 (2005), pp. 1226–1238.

[29] Mark A Hall. "Correlation-based feature selection for machine learning". PhD thesis. The University of Waikato, 1999.

[30] Stuart Russell and Peter Norvig. "Artificial Intelligence: A modern approach". In: *Artificial Intelligence. Prentice-Hall, Englewood Cliffs* 25 (1995), p. 27.

[31] Ryszard S Michalski, Jaime G Carbonell, and Tom M Mitchell. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[32] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[33] Sotiris B Kotsiantis, Ioannis D Zaharakis, and Panayiotis E Pintelas. "Machine learning: a review of classification and combining techniques". In: *Artificial Intelligence Review* 26.3 (2006), pp. 159–190.

[34] Bernhard Scholkopf. "Support Vector Machines: A Practical Consequence of Learning Theory". In: *IEEE Intelligent systems* 13 (1998).

[35] Armin Shmilovici. "Data Mining and Knowledge Discovery Handbook". In: ed. by Oded Maimon and Lior Rokach. Boston, MA: Springer US, 2005. Chap. Support Vector Machines, pp. 257–276. DOI: 10.1007/0-387-25465-X_12.

[36] Roemer Vlasveld. *Introduction to One-class Support Vector Machines*. Blog. 2013.

[37] Thorsten Joachims. *Text categorization with support vector machines: Learning with many relevant features*. Springer, 1998.

[38] Sujun Hua and Zhirong Sun. "Support vector machine approach for protein subcellular localization prediction". In: *Bioinformatics* 17.8 (2001), pp. 721–728.

[39] Li-Juan Cao and Francis EH Tay. "Support vector machine with adaptive parameters in financial time series forecasting". In: *Neural Networks, IEEE Transactions on* 14.6 (2003), pp. 1506–1518.

[40] Bernhard Schölkopf et al. "Support Vector Method for Novelty Detection." In: *NIPS*. Vol. 12. Citeseer. 1999, pp. 582–588.

[41] Cynthia Wagner, Jérôme François, Thomas Engel, et al. "Machine learning approach for ip-flow record anomaly detection". In: *NETWORKING 2011*. Springer, 2011, pp. 28–39.

[42] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. "A practical guide to support vector classification". In: (2003).

[43] Hsuan-Tien Lin and Chih-Jen Lin. "A study on sigmoid kernels for SVM and the training of non-PSD kernels by SMO-type methods". In: *submitted to Neural Computation* (2003), pp. 1–32.

[44] Ron Kohavi et al. "A study of cross-validation and bootstrap for accuracy estimation and model selection". In: *Ijcai*. Vol. 14. 2. 1995, pp. 1137–1145.

[45] Vern Paxson. "Bro: a system for detecting network intruders in real-time". In: *Computer networks* 31.23 (1999), pp. 2435–2463.

[46] Dat Tran et al. "Fuzzy vector quantization for network intrusion detection". In: *Granular Computing, 2007. GRC 2007. IEEE International Conference on*. IEEE. 2007, pp. 566–566.

[47] Emeric Nasi. *Bypass Antivirus Dynamic Analysis: Limitations of the AV model and how to exploit them*. 2014.

[48] James P Anderson. *Computer security threat monitoring and surveillance*. Tech. rep. Technical report, James P. Anderson Company, Fort Washington, Pennsylvania, 1980.

[49] Terran Lane and Carla E Brodley. "An application of machine learning to anomaly detection". In: *Proceedings of the 20th National Information Systems Security Conference*. Vol. 377. Baltimore, USA. 1997, pp. 366–380.

[50] Monowar H Bhuyan, Dhruba Kumar Bhattacharyya, and Jugal Kumar Kalita. "Network anomaly detection: methods, systems and tools". In: *Communications Surveys & Tutorials, IEEE* 16.1 (2014), pp. 303–336.

[51] Zheng Zhang et al. "HIDE: a hierarchical network intrusion detection system using statistical preprocessing and neural network classification". In: *Proc. IEEE Workshop on Information Assurance and Security*. 2001, pp. 85–90.

[52] Anup K Ghosh and Aaron Schwartzbard. "A Study in Using Neural Networks for Anomaly and Misuse Detection." In: *USENIX Security*. 1999.

[53] Guisong Liu, Zhang Yi, and Shangming Yang. "A hierarchical intrusion detection model based on the PCA neural networks". In: *Neurocomputing* 70.7 (2007), pp. 1561–1568.

[54] Ugo Fiore et al. "Network anomaly detection with the restricted Boltzmann machine". In: *Neurocomputing* 122 (2013), pp. 13–23.

[55] Cuixiao Zhang, Guobing Zhang, and Shanshan Sun. "A mixed unsupervised clustering-based intrusion detection model". In: *Genetic and Evolutionary Computing, 2009. WGEC'09. 3rd International Conference on*. IEEE. 2009, pp. 426–428.

[56] Kingsly Leung and Christopher Leckie. "Unsupervised anomaly detection in network intrusion detection using clusters". In: *Proceedings of the Twenty-eighth Australasian conference on Computer Science-Volume 38*. Australian Computer Society, Inc. 2005, pp. 333–342.

[57] Martin Roesch et al. "Snort: Lightweight Intrusion Detection for Networks." In: *LISA*. Vol. 99. 1. 1999, pp. 229–238.

[58] Srinivas Mukkamala, Guadalupe Janoski, and Andrew Sung. "Intrusion detection using neural networks and support vector machines". In: *Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on*. Vol. 2. IEEE. 2002, pp. 1702–1707.

[59] Francesco Palmieri et al. "On the detection of card-sharing traffic through wavelet analysis and Support Vector Machines". In: *Applied Soft Computing* 13.1 (2013), pp. 615–627.

[60] Ke Wang, Janak J Parekh, and Salvatore J Stolfo. "Anagram: A content anomaly detector resistant to mimicry attack". In: *Recent Advances in Intrusion Detection*. Springer. 2006, pp. 226–248.

[61] Christopher Kruegel, Giovanni Vigna, and William Robertson. "A multi-model approach to the detection of web-based attacks". In: *Computer Networks* 48.5 (2005), pp. 717–738.

[62] Yangxin Wang, Johnny Wong, and Andrew Miner. "Anomaly intrusion detection using one class SVM". In: *Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC*. IEEE. 2004, pp. 358–364.

[63] C. Edwards, O. Kharif, and M. Riley. "Human Errors Fuel Hacking as Test Shows Nothing Stops Idiocy". In: *Zugriff am* 3 (2011).

[64] Vinoo Thomas, Prashanth Ramagopal, and Rahul Mohandas. "The rise of autorun-based malware". In: *McAfee Avert Labs., McAfee Inc* (2009).

[65] Ben Gottesman. *U3*. PC Magazine, 2005.

[66] Hak5. *USB Switchblade*. 2016.

[67] BlackHat. *BadUSB - On Accessories that Turn Evil by Karsten Nohl + Jakob Lell*. Youtube, 2014.

[68] Paul Ducklin. *BadUSB – now with Do-It-Yourself instructions*. 2014.

[69] Donald Rich. "Authentication in transient storage device attachments". In: *Computer* 40.4 (2007), pp. 102–104.

[70] Microsoft. *About Enhanced Storage*. 2016.

[71] Kingston. *Kingston Digital Ships 960GB Business-Class SSD*. 2015.

[72] Jon Coulter. *Crucial MX200 500GB SSD Review*. 2015.

[73] Les Tokar. *Seagate SandForce SF3500 On Display as Seagate Moves SandForce in a New Direction – Computex 2015 Update*. 2015.

[74] Kristian Vättö. *The Samsung SSD 850 EVO mSATA/M.2 Review*. 2015.

[75] Brian Westover. *IronKey Personal S250 16GB Secure Drive*. 2013.

[76] Dave Jing Tian, Adam Bates, and Kevin Butler. "Defending Against Malicious USB Firmware with GoodUSB". In: *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM. 2015, pp. 261–270.

[77] Microsoft. *Microsoft Security Bulletin MS16-033 - Important*. 2016.

[78] Pete Zaitcev. "The usbmon: USB monitoring framework". In: *Linux Symposium*. 2005, p. 291.

[79] *Wireshark: Go Deep.* Webpage. Wireshark Foundation.

[80] *Introduction to One-class Support Vector Machines*. Code library. Core Security, 2007.

[81] Yu-Fang Zhang, Zhong-Yang Xiong, and Xiu-Qiong Wang. "Distributed intrusion detection based on clustering". In: *Machine Learning and Cybernetics, 2005. Proceedings of 2005 International Conference on*. Vol. 4. IEEE. 2005, pp. 2379–2383.

[82] Pedro Casas, Johan Mazel, and Philippe Owezarski. "Unsupervised network intrusion detection systems: Detecting the unknown without knowledge". In: *Computer Communications* 35.7 (2012), pp. 772–783.

[83] I Hareesh et al. "Anomaly detection system based on analysis of packet header and payload histograms". In: *Recent Trends in Information Technology (ICRTIT), 2011 International Conference on*. IEEE. 2011, pp. 412–416.

[84] Yuji Waizumi et al. "Distributed early worm detection based on payload histograms". In: *Communications, 2007. ICC'07. IEEE International Conference on*. IEEE. 2007, pp. 1404–1408.